Spring 2019

Modeling a chaotic billiard: The Bunimovich Stadium

Randal Shoemaker

Modeling a Chaotic Billiard: The Bunimovich Stadium

An Honors College Project Presented to

the Faculty of the Undergraduate

College of Integrated Science and Technology

James Madison University

by Randal W. Shoemaker

April 2019

Accepted by the faculty of the Department of Computer Science, James Madison University, in partial fulfillment of the requirements for the Honors College.

FACULTY COMMITTEE:

Project Advisor: Ramon A. Mata-Toledo, Ph.D.
Professor, Computer Science

Reader: David H. Bernstein, Ph.D.
Professor, Computer Science

Reader: James S. Sochacki, Ph. D.
Professor, Mathematics

Reader: Bryce Weaver, Ph. D.
Research Scientist, Mathematics

HONORS COLLEGE APPROVAL:

Bradley R. Newcomer, Ph.D.,
Dean, Honors College

PUBLIC PRESENTATION

This work is accepted for presentation, in part or in full, at JMU CS Dept research seminar on 04/12/19.

This work is dedicated to the memory of my grandfather Warren A. Shultz. I also dedicate it to my grandmother Loretta, my wife Eliza, and son, Rhyan.

Contents

List of Figures

Acknowledgments

I would like to thank my advisor, Professor Ramon Mata-Toledo, for his tireless efforts in making sure this work is of the highest quality. I also want to thank Ramon for encouraging me to pursue a senior honors thesis, without his encouragement this work would not have been possible. I would like to thank my readers, Professors Jim Sochacki and David Bernstein, and Bryce Weaver for all of their advice and suggestions. Their suggestions have enabled me to greatly improve this work. I thank Bryce for introducing me to this subject and helping me to understand the ideas necessary for this work. I would also like to thank Professor Thomas Crow from my previous institution, Blue Ridge Community College, for preparing me to succeed in the mathematical realm. I would also like to thank Professors Franklin, Watkins, Phillips, Maxfield, Ryan, and Lewis of Blue Ridge for preparing me to succeed at the University level. Thanks also to James Madison University's Honors College, Computer Science Department and for allowing me this opportunity.

Abstract

The Bunimovich stadium is a chaotic dynamical system in which a single particle, known as a billiard, moves indefinitely within a barrier without loss of momentum. Mathematicians and physicists have been interested in its properties since it was discovered to be chaotic in the 1970's [5] [3] [4]. The Bunimovich stadium is actively researched [9]. This thesis and its accompanying software, the *Bunimovich Stadia Evolution Viewer* (BSEV), present a novel visual representation of the the chaotic dynamical system. The goal for the software is to provide insights into the stadium's properties to aid researchers. This tool allows one to visualize the system's evolution and see how sets of close billiard trajectories diverge over time. Current tools only provide views of the billiard moving within the boundary. This tool provides a view of the system in terms of the set of all possible collisions, known as the collision space. The collision space of the Bunimovich stadium is equivalent to a cylinder and the new tool presents the visualizations on the cylinder. This view shows the evolution of nearby billiard trajectories in the collision space. Such visualizations may provide insights into an important aspect of the stadium's chaos, its sensitivity to initial conditions. The BSEV allows users to supply initial conditions for the system and generate images showing the system's evolution with those inputs. These images specifically aid in visualizing the system's sensitive dependence. Other viewers do not provide the ability to visualize the system in this way making it difficult for them to provide insights into the chaotic nature of the stadium. The BSEV does not have these limitations and can provide new insights into this chaotic dynamical system. The software for the BSEV can be freely downloaded from the following URL: https://github.com/shoemarw/BunimovichStadium

Chapter 1

Introduction

The goal of this thesis is to produce a tool, in the form of software, which provides a novel way of visualizing the evolution of a dynamical system called the Bunimovich stadium billiard. The Bunimovich stadium consists of a single particle, called a billiard, bouncing within a boundary without loss of momentum. The boundary is shown in Figure 1.1.

Figure 1.1: The Bunimovich stadium billiard

When the billiard encounters the boundary it bounces off with its angle of reflection equal to its angle of incidence. Our goal is to produce a tool which enables visualization of the system's evolution. Specifically we seek to visualize how slight changes to the starting conditions of the billiard affect its trajectory and where such changes have a greater effect. For an illustration of this concept consider Figure 1.2.

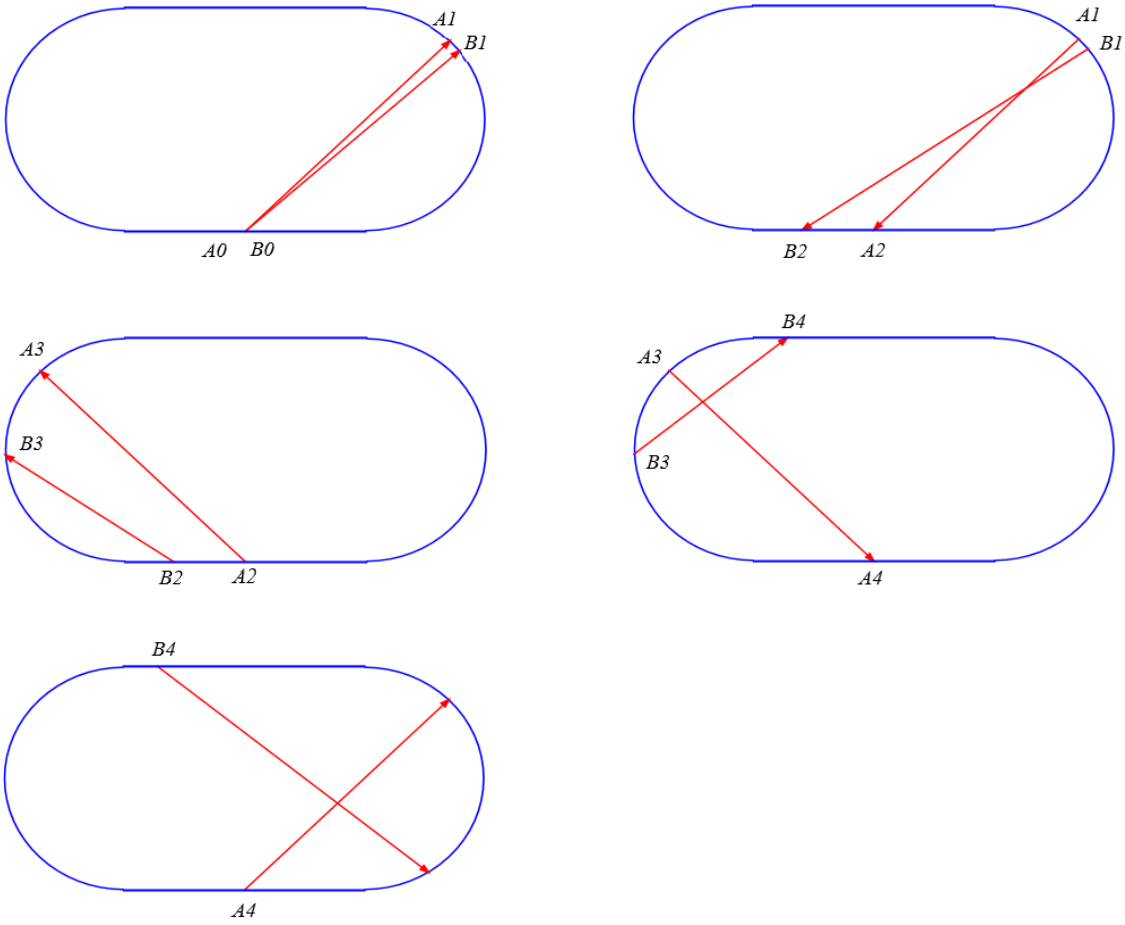

Figure 1.2: Slightly changing the initial conditions of a billiard can greatly affect its trajectory

If the billiard starts at the initial condition labeled $A0$ and aims toward $A1$ it will take the path $A0, A1, A2, A3$,-$A4, ...$ as shown in Figure 1.2. However, if the billiard starts with the slightly different initial condition labeled $B0$ and aims toward $B1$ then it will take the path $B0, B1, B2, B3, B4, ...$. Note that the trajectories of the two billiards are very different despite only having a slight difference in their initial conditions. Some questions arise: how can we visualize the difference in these trajectories are and how "fast" are these trajectories becoming "different"? If we consider the collisions as points in a space that describe where a collision occurs and where the next collision will be, then we can view how the sequences of collisions differ using the tool developed for this thesis. Considering collisions as points is traditional in the study of billiards [5].

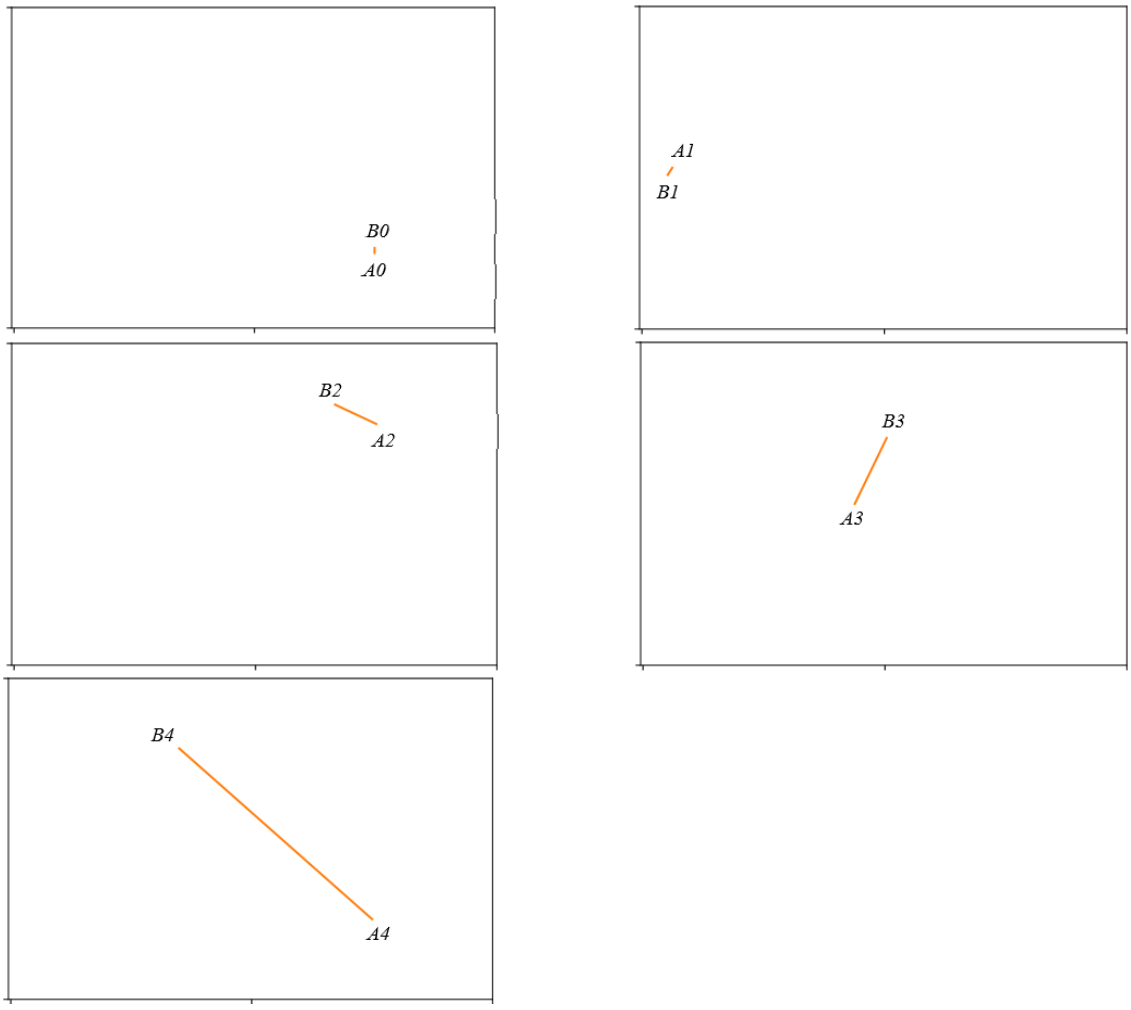

Figure 1.3: The trajectories from Figure 1.2 where the collisions are viewed as points in a space

Shown in Figure 1.3 are the same trajectories in Figure 1.2 except the collisions are viewed as points in a space. When viewing the collisions in this manner we can see how the trajectories are "different" in a new way. With this view we can see how "far" apart the trajectories have become by observing the length of the line segment connecting them. As the billiard's two trajectories become more "different" in Figure 1.2 the length of the line segment in Figure 1.3 becomes larger. This helps us see how a slight change in the initial conditions of the billiard changes the trajectory. Knowing this we can now ask: how we can visualize the speed at which the two trajectories diverge? To answer this question consider Figure 1.4 where the difference of the paths $C0, C1, C2, ...$ and $D0, D1, D2, ...$ is small. In contrast with Figure 1.2, the paths remain "close" and don't diverge, so we say the divergence exhibited in Figure 1.2 is greater than that in Figure 1.4. If we view the collisions in Figure 1.4 as points in a space we get the images in Figure 1.5. When we introduce the mathematical preliminaries for this thesis our discussion will be more formal. For now we can think of a measure of the divergence of close initial conditions in terms of the how fast the lengths of the line segments

in Figures 1.3 and 1.5 grow.

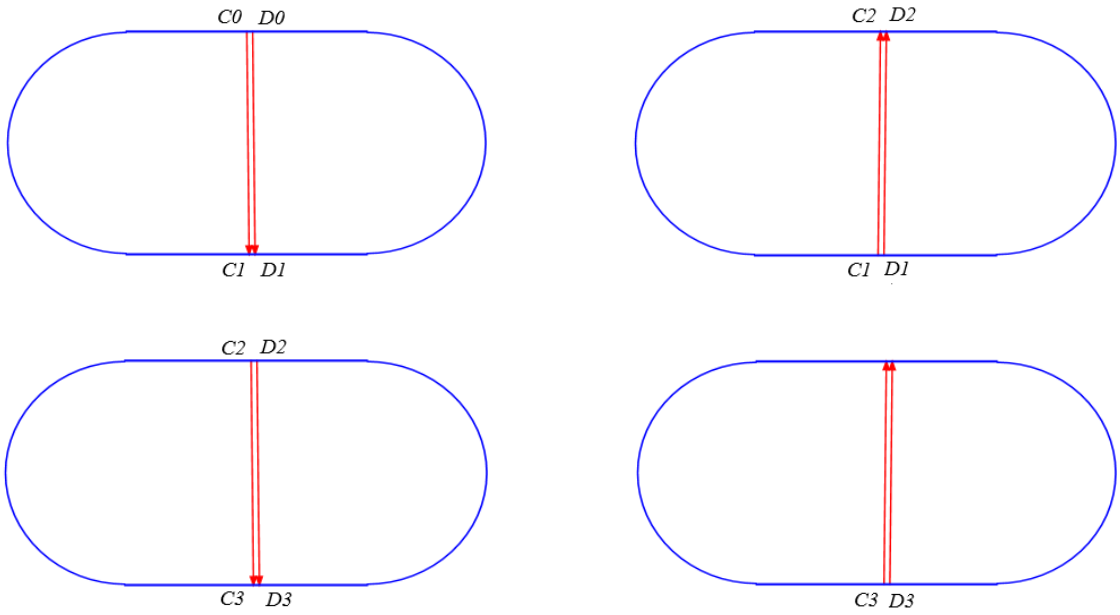

Figure 1.4: Example where two billiard paths stay close

As mentioned before, the goal of this thesis is to produce a tool that enables a *novel* visualization of the Bunimovich stadium billiard's evolution. The images in Figures 1.3 and 1.5 allow changes in trajectories to be visualized more easily than those in Figures 1.2 and 1.4. To get a better visualization it would be useful see the trajectories of more than two initially "close" billiards in the same way as provided in Figures 1.3 and 1.5. The tool created to achieve our goal is called the *Bunimvich Stadia Evolution Viewer* (BSEV) and it provides the ability to observe the evolution of many nearby initial conditions. This allows one to see how trajectories diverge and where their divergence is greater. Images produced by this software and its use are described in Chapter 5 and the software can be freely downloaded via Github from the following URL: https://github.com/shoemarw/BunimovichStadium

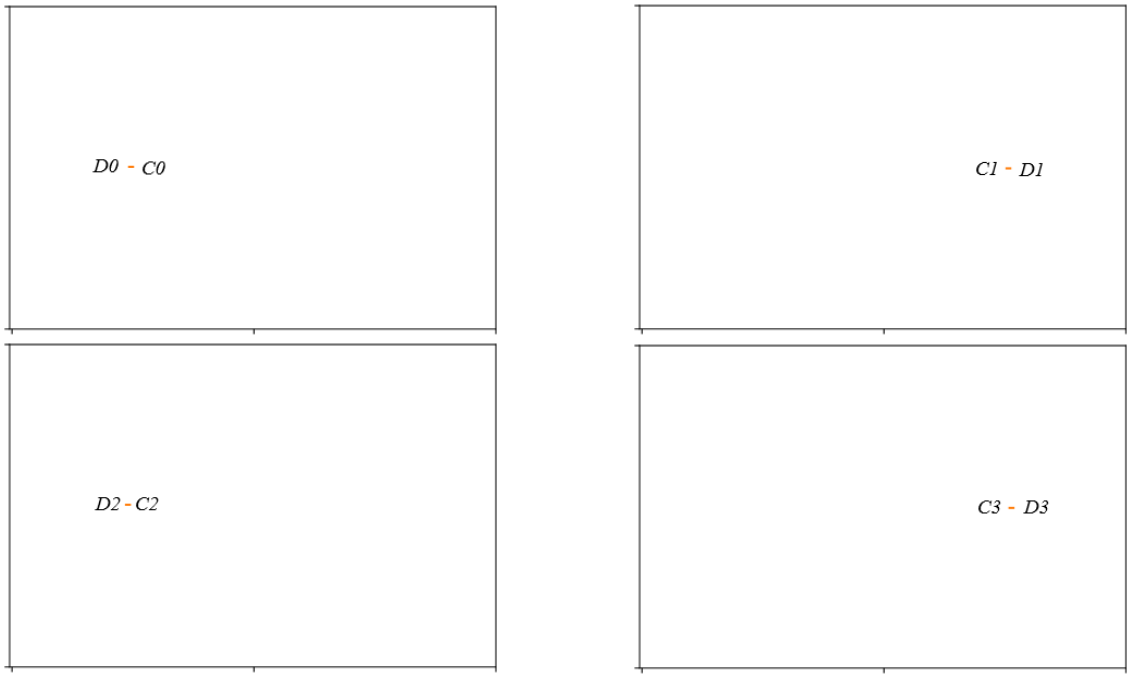

Figure 1.5: Example where two billiard paths stay close with collisions as points in a space

Current visualization tools exist for the Bunimovich stadium such as those of [2] and [10]. Both of these tools show billiards moving within the boundary and do not allow for the visualization of collisions as in the previous figures. However, this has the advantage of allowing us to visualize the billiards' trajectories between collisions. As a result some interesting behavior can be observed. For instance, using the tool from [10] one can see how billiards with initially close initial conditions become uniformly distributed throughout the stadium. This behavior can be seen in Figures 1.6 and 1.7, which contains an image of the tool from [10]. Notice that this tool does not make it easy to see the evolving relationship of paths with initially close billiards. After a few bounces it is hard to keep track of which billiards started out closer and which didn't. These tools are discussed in Chapter 3. The BSEV software can produce images where the collisions are viewed as points in a space, as in Figures 1.3 and 1.5. This means that if two billiards started out adjacent to one another the points corresponding to them in the space will be connected with a line. Therefore we can keep track of which billiards started closer to one another. We discuss the use of the improved viewer for visualizing this behavior in Chapter 5.

The Bunimovich stadium billiard is a famous mathematical object and has been studied for decades. Mathematicians and physicists are particularly interested in the behavior of this system because it is known to be chaotic. A careful discussion of billiard behavior is covered in Chapter 2. Since the stadium was discovered to be chaotic it has attracted a great deal of attention and is still an active area of research [9]. The BSEV software will help researchers because it allows one to visualize how the system evolves under

Figure 1.6: An image of an existing visualization tool for the Bunimovich stadium. Several billiards start their trajectories close together

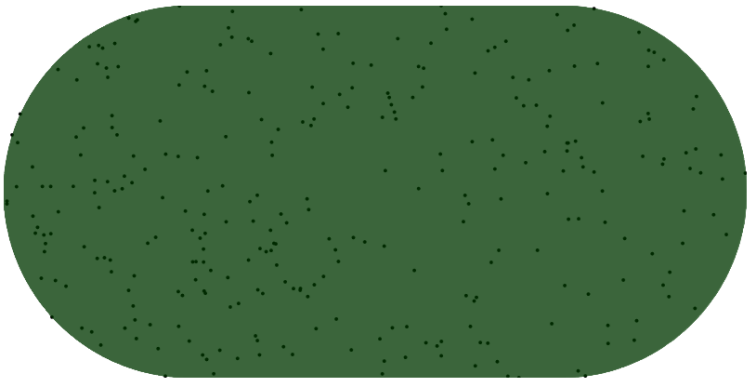

Figure 1.7: The billiards which started out close in Figure 1.6 end up approximately uniformly distributed within the boundary

slight variations of its input. Visualizing the system's evolution in this novel way could provide insights for researchers seeking to understand this chaotic system.

In the next chapter we will discuss the mathematical ideas necessary for understanding this work and the Bunimovich stadium. In Chapter 3 we discuss current visualization tools for the stadium. In Chapter 4, the software that accomplishes the goal of this thesis is presented. This software is called the "Bunimovich Stadia Evolution Viewer." In Chapter 4 we also discuss the development of the software along with its properties. In Chapter 5 we discuss some results and the novel insights provided by the BSEV. In Chapter 6 we discuss our conclusions and in Chapter 7 we discuss future venues of research with respect to this thesis.

Chapter 2

Mathematical Preliminaries

This chapter provides an introduction to the topic of dynamical systems, chaotic billiards, and specifically the Bunimovich stadium billiard. It is meant to familiarize the reader with the topics necessary to understand the work presented in this thesis and better understand its goal. The goal of this work is to develop a novel visualization tool for the Bunimovich stadium. Current visual presentations of the system make it difficult to see certain aspects of the systems evolution [10] [2]. The software developed in conjunction with this thesis, the *Bunimovich Stadia Evolution Viewer* (BSEV), aims to ameliorate this issue. To help the reader understand this goal we nust introduce the mathematical ideas key to understanding it.

2.1 Billiards and Dynamical Systems

A Bunimovich stadium is a rectangle with semi-circles capping each end, as shown in Figure 2.1. Inside this boundary, we will assume that a point particle travels in straight paths with a velocity of constant magnitude, meaning that its speed remains unchanged. The particle, known as a billiard, moves within the boundary indefinitely without friction acting on it. When the billiard encounters the boundary it bounces back with an angle of reflection equal to its angle of incidence as indicated in Figure 2.1.

During any collision, only the direction of the velocity of the billiard is changed. As stated in *Chaotic Billiards* [5], the Bunimovich stadium billiard was introduced by the Mathematician Leonid Bunimovich in 1974 [3]. The stadium is a special case of a more general class of dynamical systems called chaotic billiards first presented by Y. G. Sinai in 1970 [11]. The stadium billiard has parameters ℓ and r as shown in Figure 2.2. We define ℓ as half the length of one of the flat sides and r as the radius of one of the caps. We let $\lambda = \frac{2\ell}{r}$ and usually refer to λ as characterizing a particular Bunimovich stadium billiard table. In this work we adopt the convention of setting $r = 1$ so that $\lambda = 2\ell$, thus making λ the length of one of the

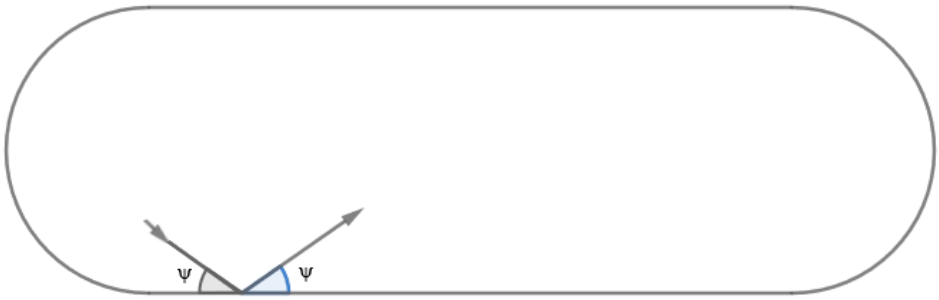

Figure 2.1: The angle of incidence (*in grey*) and angle of reflection (*in blue*) are equal in a Bunimovich stadium billiard system

flat sides. Setting $r = 1$ does not change the dynamics of the system; we discuss dynamics below.

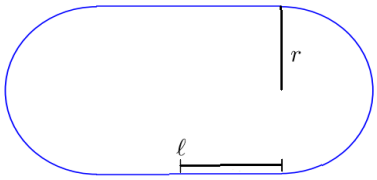

Figure 2.2: The parameters r and ℓ in a Bunimovich stadium billiard

Informally, a dynamical system is an iterated, or repeated, process which occurs over time. For the Bunimovich stadium system the associated iterated process is the continuous collision of the billiard against the boundary. The rules governing iterated processes are referred to as the **dynamics**. The **phase space** of a dynamical system is the set of all possible states that the system can be in. For example, if the system of concern consisted of only two coins, a possible state of the system could be the first coin facing heads up and the second facing tails up, this can be represented by (H, T). In this example the set of all possible states is $\{(H, H), (H, T), (T, H), (T, T)\}$. For the Bunimovich stadium system the phase space is the set of all possible ordered pairs where the first and second coordinates are the position and the direction of the billiard respectively. We assumed earlier that the speed, or magnitude of the velocity, is constant. Therefore only the direction of motion is important. Therefore an element of the phase space is an ordered pair, denoted by $(q, \hat{v})$, consisting of a position within the billiard table q and the direction at that instant $\hat{v}$. Thus, the phase space Ω is the set of all these ordered pairs.

Billiard systems, for the purposes of this work, are a class of dynamical systems in which a billiard moves within some closed boundary consisting of some combination of lines and curves connected end to end. The area within the boundary along with the boundary itself is known as the "**billiard table**" and it is denoted[1] $\mathcal{D}$. The **boundary** is denoted $\partial\mathcal{D}$. The dynamics, or billiard motion, of billiard systems can be described

[1]We shall note here that we adopt the notation from *Chaotic Billiards* [5]. This notation is typical of research concerning billiard systems.

by a function. This function takes two inputs: a position-velocity pair $(q, \hat{v})$ and a time, t, represented by a real number. The output is a new position-velocity pair. This pair represents the position and velocity of the particle after the time t. These functions are called **flows** and are usually denoted Φ^t for a given time t. To formally describe Φ^t we write $\Phi^t : \Omega \to \Omega$, which means that Φ^t takes an element of Ω and maps it to, or associates it with, an element of Ω. This is a formal way of saying that Φ^t takes a billiard moving within the table as input. It then tells us where the billiard will be and its direction after a specified amount of time t. We let $q(t)$ denote the position of a particular billiard at time t and $\hat{v}(t)$ denote the velocity at that time. For a particular billiard with an initial state (q_0, v_0) we can write $\Phi^t((q_0, v_0)) = (q(t), v(t))$ with $\Phi^0((q_0, v_0)) = (q(0), v(0)) = (q_0, v_0)$; this means that at time $t = 0$ the function Φ outputs the initial state. Later we shall introduce the discrete analog of Φ^t which only considers collisions.

2.2 The Behavior of Dynamical Systems

2.2.1 Sensitivity to Initial Conditions

Recall the goal of this work is to present a novel visual representation of the the evolution of the Bunimovich stadium billiard. Our model specifically aims to provide a visualization which aides in observing how and where the system is *sensitivite to initial conditions*. We now describe this property of dynamical systems. A dynamical system such as a billiard system can be chaotic which, in this context, has a very precise meaning. Chaos in these systems can be thought of as the general difficulty of precisely predicting any future state of the billiard system given an initial position and velocity. To understand why such a difficulty arises consider the billiard table in Figure 2.3. In what follows we discuss the properties a dynamical system must have in order to be called chaotic. These properties are *sensitivity to initial conditions*, *density of periodic orbits*, and *transitivity*.

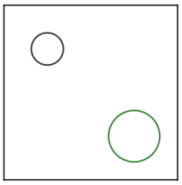

Figure 2.3: A billiard table.

In Figure 2.3, we have a square billiard table that contains two circles. Billiards moving within this system can bounce off of the sides of the square or the perimeter of either circle. Figure 2.4 shows two possible billiard paths with the same starting position and slightly different directions. The first billiard starts at the point labeled A in Figure 2.4 traveling in such a direction so that it strikes the smaller circle at

point B. It then continues on, striking the larger circle and, finally, it hits the boundary at point D. For the second path, a billiard starts at point A aiming in almost the same direction as before, but this time striking the smaller circle at point C. This billiard would eventually reach point E after striking the larger circle.

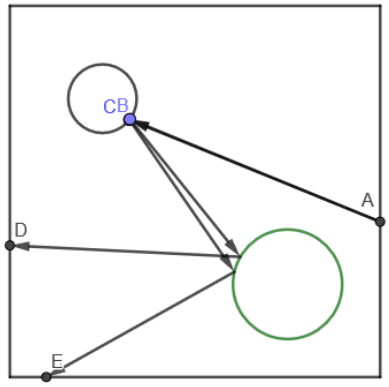

Figure 2.4: Two possible paths of a billiard in the table from Figure 2.3.

Notice that in Figure 2.4 it looks as if both billiards have collided with the smaller circle at the same location. In Figure 2.5 we have the same billiard table and billiard paths as Figure 2.4 except we have zoomed in near points B and C to show that they are not the same.

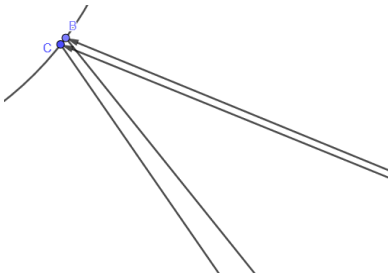

Figure 2.5: This is the same image as Figure 2.4 except zoomed in around points B and C.

Notice that the billiard paths shown in Figures 2.4 and 2.5 start at the same position, A, with only a barely noticeable difference in direction as shown in Figure 2.5. Of interest here is that they have ended up in very different locations after bouncing off the larger circle. Indeed, if one were to witness their motion just after their collisions with points D and E, one would be unable to predict easily that they started in almost the same state. The behavior exhibited above is called **sensitivity to initial conditions**, and it is related to the previously mentioned difficulty of predicting the future state of a chaotic dynamical system. The goal of this research is to enable the visualization of sensitive dependence in the Bunimovich stadium billiard. We must note that sensitivity to initial conditions is not equivalent to chaos; it is only one of the requisite behaviors for a dynamical system to be labeled chaotic.

2.2.2 Orbits and Density

Another requisite behavior for chaos in dynamical systems and in particular, billiards, concerns the idea of periodic orbits. Before discussing periodic orbits we must first discuss trajectories. The **trajectory** of a billiard is the path it takes from its initial starting point onward. Since the billiard moves indefinitely, the trajectory goes on indefinitely. A trajectory is composed of links [5]. A **link** is the line segment connecting any two successive collisions of a billiard with the boundary $\partial\mathcal{D}$. This can be seen in Figure 2.6. Thus a

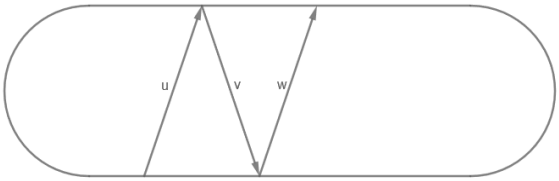

Figure 2.6: Part of a trajectory with 3 links. The links are labeled u,v, and w.

trajectory can be represented as a sequence of consecutive links.

Dynamical systems can be simplified to make them "easier" to study. The key here is to note that each link can be identified with the collision preceding it. Notice that for each link there is exactly one collision preceding it and for each collision there is exactly one link after it. This yields a natural one-to-one correspondence between links and collisions. In order to properly identify a collision with a link, we need to keep track of more than the position where the collision takes place; we must also keep track of the direction of the billiard after the collision; this value is the direction of the post-collision velocity. Let s denote the position on the boundary ∂D where the collision takes place and let ψ denote the direction. This yields a position-direction ordered pair (s, ψ); an example of this pair can be seen in Figure 2.7. Formally, ψ is the

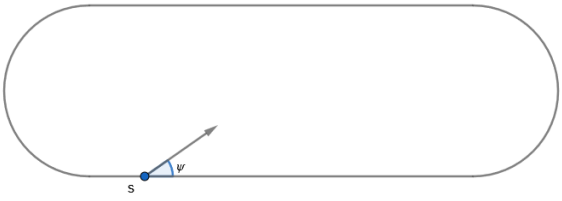

Figure 2.7: A position-direction ordered pair on a side (s, ψ).

counter-clockwise measured angle between the post-collision velocity and the line tangent to the boundary $\partial\mathcal{D}$ at the point of collision s. Notice that in Figure 2.7 s is on a flat side of the stadium, so the tangent line coincides with the flat side. When s is on a cap we measure ψ as seen in Figure 2.8. We call the set of all such pairs the collision space and denote it as $\mathcal{M}$. Recall that the trajectory is the sequence of consecutive links. Therefore, associated with each possible billiard trajectory is a sequence of collisions because each

Figure 2.8: A position-direction ordered pair on a cap (s, ψ).

collision corresponds to a link. Now we can discuss orbits and periodic orbits.

An **orbit**, in the context of billiard systems and, in particular, of the Bunimovich stadium, is the sequence of collisions with the boundary $\partial\mathcal{D}$ in the order in which they occur. As previously mentioned, this sequence of collisions corresponds to a billiard trajectory. One can think of the billiard starting its motion and then colliding with point A on the boundary, bouncing off, and then colliding with another point B on the boundary. The billiard could then collide with point C and so forth. This is illustrated in Figure 2.9.

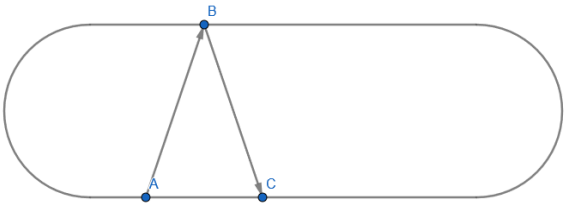

Figure 2.9: An example showing part of an orbit

The orbit would then be $A, B, C, ...$ Recall earlier that each collision is associated with a position-direction pair (s, ψ). Thus the points A, B, and C are actually position-direction pairs. The **periodic orbit** would be the part of a sequence that repeats itself.

Figure 2.10: An example of a periodic orbit

For a simple example think about a billiard that starts at point A in Figure 2.10. This time, lets say it bounces straight across so that its angle incidence and reflection are equal to $\frac{\pi}{2}$, then lets call the point on the boundary where it collides on the opposite side point D. The orbit of this billiard would then be $\overline{AD}$ because it would bounce back and forth between the points A and D indefinitely. The **periodic orbit**, or the orbit which repeats, would be A, D, and we call A and D the **periodic points**. Periodic points are

important in the study of dynamical systems. However, before we can discuss how periodic points relate to chaotic behavior we must introduce a new concept, density.

We say that for a dynamical system to be called chaotic the set of periodic points must be dense in the phase space. In the context of billiards this translates as: the set of periodic points must be dense in the collision space $\mathcal{M}$. The later can then be interpreted as: given any position-direction pair p in $\mathcal{M}$, p is either a point on a periodic orbit or there is a periodic point "arbitrarily close" to p. We must now discuss what is meant by "arbitrarily close".

In the context of billiards we say that the periodic points are dense in the collision space $\mathcal{M}$ precisely when given any $p \in \mathcal{M}$ and any $\epsilon > 0$ we can find a periodic point $q_\epsilon \in \mathcal{M}$ with the distance between p and q_ϵ less than ϵ. Notice that if p is a periodic point we can set $q_\epsilon = p$ and the distance between the two points would be 0, which is less than ϵ, thus satisfying the condition. If the above condition is satisfied, we would then be justified in saying that p has a periodic point "arbitrarily close" to it. To make this notion precise we must define what is meant by "the distance between two collision points," however we shall put off this discussion until we formally develop the collision space $\mathcal{M}$ in section 1.3.3. For now we shall think of collision points as being "close" when they are close in both position and direction. For an example consider the position-direction pair labeled J in Figure 2.11.

Figure 2.11: A position-direction pair in the collision space $\mathcal{M}$

An example of a "close" periodic point in the collision space $\mathcal{M}$ is the point labeled E in Figure 2.12, in this image we can see that E is a periodic point.

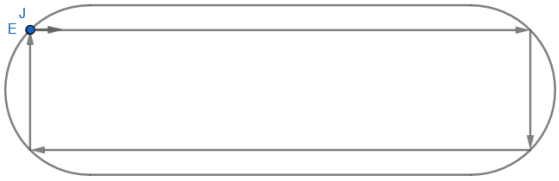

Figure 2.12: E is a periodic point.

If we zoom into Figure 2.12, as shown in Figure 2.13, we can see that indeed the two points are "close" in both position and direction. They are close in position because they hit the boundary $\partial\mathcal{D}$ at almost the

same point. They are also close in direction because they leave $\partial\mathcal{D}$ going in almost the same direction. We shall make this notion more precise in section 1.3.3 during our development of $\mathcal{M}$.

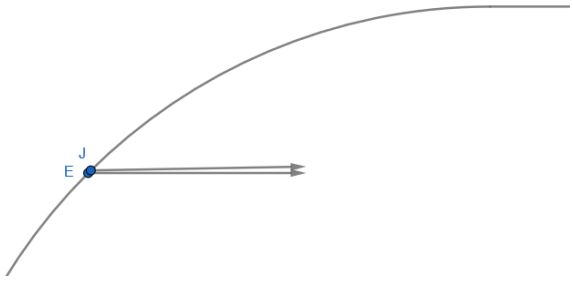

Figure 2.13: Zooming into Figure 2.12 reveals that E is "close" to J.

If any point in the phase space is either a periodic point or is "arbitrarily close" to a periodic point, then the density condition is met. So the **density** condition is met if given any collision point p, p is a periodic point or given any positive distance ϵ we can find a periodic point q_ϵ where the distance between p and q_ϵ is less than ϵ. We formally state this as

Definition 2.1. *For every $p \in \mathcal{M}$ and for every $\epsilon > 0$ there is a periodic point q such that $d(p,q) < \epsilon$. Where $d(p,q)$ denotes the distance between p and q, which will be defined in Section 2.3.3.*

We must note that having *sensitivity to initial conditions* and *periodic points dense in the phase space* is still not sufficient for a dynamical system to be chaotic. There is one more condition that needs to be met for the label of chaos to be bestowed upon a dynamical system, including billiards.

2.2.3 Transitivity and Chaos

The final condition is called transitivity. The idea of **transitivity** in a billiard system can be stated as follows[2]. Given any two points in the collision space, we can always find a third point which orbits as "close" as we like to both of the other points. The orbit of the third point would contain two points in the collision space $\mathcal{M}$ that come "arbitrarily close" to the original two points respectively. So we define **Transitivity** as

Definition 2.2. *For every $p, q \in \mathcal{M}$ and $\epsilon > 0$ there is an $r \in \mathcal{M}$ that contains points p', q' in its orbit, where $d(p,p') < \epsilon$ and $d(q,q') < \epsilon$. Where $d(p,p')$ denotes the distance between p and p', which we shall formally define in 2.3.3.*

[2]Another definition of transitivity uses open sets: For every pair of open sets U V in the phase space, there exists $x \in U$ and $n \in \mathbb{Z}^+$ such that $F^n(x) \in V$. This definition is used in [1]. We avoid this definition because we have no need of defining opens sets, so instead we use the definition stated above.

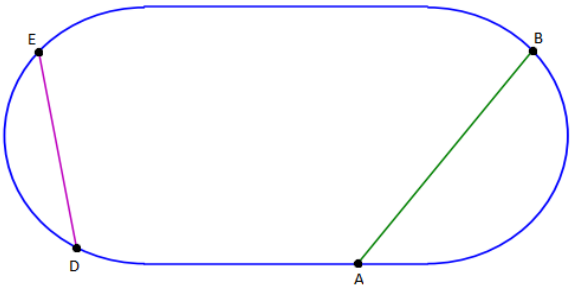

Figure 2.14: Two points, AB and DE in the collision space.

For example, suppose we are given the collision points in Figure 2.14. The first collision-point AB starts at the location on the boundary $\partial\mathcal{D}$ labeled A and goes toward the location labeled B. The second collision-point DE starts at the location labeled D and goes toward the location labeled E. When given two such points in a transitive dynamical system there is always a third point whose orbit comes arbitrarily "close" to the original two points respectively; in Figure 2.15 we can see an example of such a point. The collision-point AC starts at location A with a direction very close to that of AB. The billiard traveling along the link corresponding to AC then collides with $\partial\mathcal{D}$ at location C and then collides with it again at location D, where it has a direction close to that of DE; the billiard finally collides with location F. From Figure 2.15 we can see that the orbit of AC comes "close" to the collision points corresponding to AB and DE. So for a particular $\epsilon > 0$ in the definition of transitivity the point AC has the points AC and DE in its orbit that are within ϵ distance of AB and DE respectively. For the billiard system to be transitive we would need to be able to find such points for any given pair of points and any ϵ.

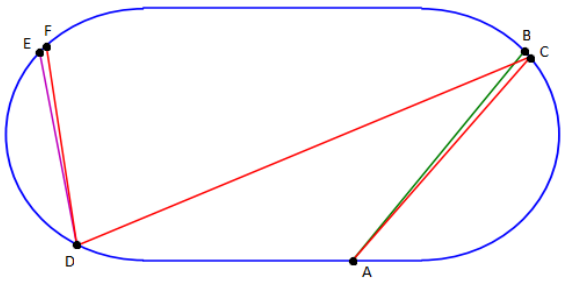

Figure 2.15: Example of transitivity

We now have defined the concepts necessary to define a chaotic dynamical system. While chaos does not have a definition which is universally agreed upon, the one we provide is generally accepted. Following Devaney [6] we define a chaotic dynamical system as follows:

Definition 2.3. *A dynamical system is* ***chaotic*** *if it has the following properties:*

(1) Sensitivity to initial conditions.

(2) The set of periodic points is dense in the phase space.

(3) The system is transitive.

As mentioned before, this is not the only definition of chaos that is used in the field of Dynamical Systems. There is another definition which only uses the last two properties. In fact, if the function describing the dynamics is continuous, the last two properties imply sensitive dependence [1]. For the Bunimovich stadium billiard, the function describing the dynamics turns out to be continuous and we discuss this fact Section 2.4.

Since we have introduced the properties which define chaos we can turn our discussion toward chaotic billiards.

2.3 Chaotic Billiards

2.3.1 The components of $\partial\mathcal{D}$

When billiard systems are discussed it is useful to be able to describe certain properties of the pieces that make up the boundary $\partial\mathcal{D}$. For billiard systems it is assumed that $\partial\mathcal{D}$ consists of a finite number of pieces such as lines, curves, and segments of circles. Suppose there are n such pieces, we denote each piece Γ_i for $i \leq n$ and refer to it as a **wall**. For instance, in the Bunimovich stadium we label the right cap Γ_1, the top side Γ_2, the left cap Γ_3 and the bottom side Γ_4. Then the boundary $\partial\mathcal{D}$ results from the juxtaposition of the walls appropriately as in Figure 2.16.

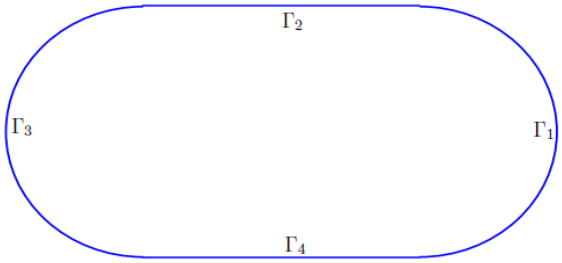

Figure 2.16: The walls of the stadium billiard.

Definition 2.4. *The walls Γ_i of billiard tables are classified in the following way [5]:*

1. *A given Γ_i is **flat** if it is a line segment.*
2. *A given Γ_i is **focusing** if the curve is convex to the interior.*
3. *A given Γ_i is **dispersing** if the curve is concave to the interior.*

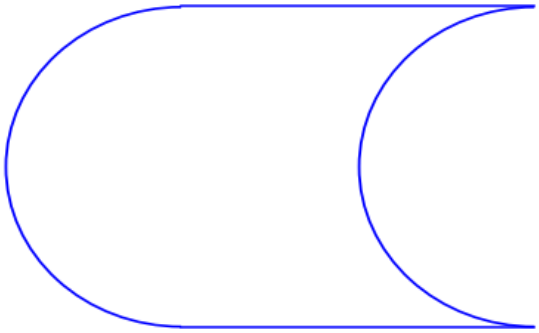

Figure 2.17: Example of focusing (left), flat (top and bottom), and dispersing (right) walls.

An example of a dispersing wall can be seen in the billiard table in Figure 2.17. The curve on the right is dispersing because billiards moving in the table only strike the part that is concave to the interior. Dispersing walls are named this way because when close billiards hit this kind of wall they are dispersed. The walls making up the top and bottom of the billiard table in Figure 2.17 are examples of flat walls. Examples of focusing walls are the caps of the Bunimovich stadium. They are focusing because a billiard can only strike the inside of the circle and thus become 'focused' inward. If each wall of a billiard table is dispersing, the table is referred to as **dispersing** or **everywhere dispersing**. In 1970 Yakov G. Sinai showed that dispersing billiards are chaotic[11]. Billiard tables that have no dispersing walls are called **nowhere dispersing** or **focusing** tables. For a period after Y. G. Sinai showed that dispersing billiards were chaotic, they were the only known class of chaotic billiards [5]. In 1974 Lenoid Bunimovich, a student of Y. G. Sinai, came up with a set of billiard systems that contained only flat and focusing walls. He showed that billiards of this type were chaotic in [3] and they came to be known as Bunimovich billiards. It was a very surprising result because it was previously thought that only systems with dispersing components were chaotic [5]. The Bunimovich stadium is a type of Bunimovich billiard so L. A. Bunimovich's proof also yielded the surprising result that the stadium is also chaotic [4]. It can be shown that billiard tables made of only polygons, circles, or ellipses are not chaotic. Notice that the stadium billiard becomes a circle when the lengths of the flat walls are zero. As long as the flat walls are not zero, or even very small, the system is chaotic. When the length of the flat walls becomes zero the system is no longer chaotic because circular

tables aren't chaotic. The fact that this simple system can exhibit chaotic behavior is a leading reason why it became popular among mathematicians and physicists after Bunimovich's publications in 1974 and 1979 [5].

2.3.2 Quantifying Sensitive Dependence

One of the most important aspects of the systems's chaos, it terms of this work, concerns sensitivity to initial conditions. When discussing chaotic dynamical systems it is useful to quantify this behavior to precisely describe the sensitivity. The value associated with this quantification is called a Lyapunov exponent. **Lyapunov exponents** are a measure of how quickly trajectories diverge, or grow apart, over a large period of time. The Lyapunov exponents associated with a point in the phase space measure how fast nearby trajectories diverge. For example recall Figure 2.4 on page 19 and consider the point representing the position-direction pair starting at position A and ending up at position D. If we look at "nearby" position-direction pairs we can see that their trajectories diverge quickly. For instance, observe the pair starting at position A and ending up at position E. After 3 collisions the resulting pairs are distant because they differ in position and direction. So we can imagine that over a large period of time the trajectories would be vastly different. The Lyapunov exponents for this point would be large. If Lyapunov exponents measure how quick trajectories diverge over long time scales, what about when they are zero? When the Lyapunov exponents are zero this indicates that the divergence is very slow for long periods of time. When a point in the phase space of a dynamical system has non-zero Lyapunov exponents we say the point is hyperbolic. Recall that the dynamics of a billiard system describes the motion of a billiard over time. We say that a billiard system is **hyperbolic** if almost every point in the phase space is hyperbolic, which means that the the points that aren't hyperbolic are negligible; in the sense that the probability of a randomly chosen point being non-hyperbolic is zero. When Y. G. Sinai showed that everywhere dispersing billiard tables were chaotic, the hyperbolicity[3] of these systems was one of the facts he established [11].

In his 1974 paper, Bunimovich, showed that some "nowhere dispersing" billiards were hyperbolic [3]. Recall that a "nowhere dispersing" billiard table is defined as one where there are no dispersing walls; so the table only consists of focusing and flat walls. Also recall that the class of billiard tables with only focusing and flat walls is referred to as Bunimovich billiards and the stadium is a particular instance of this class. We stated that a dynamical system is hyperbolic if almost every point in its phase space is hyperbolic, so is the stadium hyperbolic? In the stadium, there are trajectories which are not hyperbolic; these are referred to as type B trajectories [5]. Specifically, **type B** trajectories are trajectories that only collide with the flat

[3]It must be noted that there are different classifications of hyperbolic systems: uniformly and non-uniformly hyperbolic systems. In this text when we use the term hyperbolic, we mean non-uniformly hyperbolic. The distinction is beyond the scope of this work. For more information we direct the reader to [5].

boundary components, for an example refer to Figure 2.10. It turns out that if the probability of choosing a point with a type B trajectory is zero then the system is hyperbolic [5]; this condition is met by the stadium billiard [5].

2.3.3 The Collision Space $\mathcal{M}$

Recall that the **collision space** $\mathcal{M}$ of a Bunimovich stadium is the set of all position-direction ordered pairs (s, ψ). We now provide a careful definition of ψ. First we must describe the inward normal and tangent vectors of a billiard table.

Figure 2.18: The normal $\hat{N}$ and tangent $\hat{T}$ to a point on $\partial\mathcal{D}$

For each point on the boundary of the Bunimovich table there is an inward normal vector. For a point on one of the flat walls, the **inward normal** vector associated with that point is perpendicular to the wall and points toward the inside of the table. For a point on one of the focusing walls, the associated inward normal points toward the center of the semi-circle as shown in Figure 2.18. The **tangent** vector for any point on the boundary $\partial\mathcal{D}$ is perpendicular to the associated inward normal; in addition, as we travel around $\partial\mathcal{D}$ in a counter-clockwise direction, the tangent always points in the direction we are traveling as seen in Figure 2.18. Notice that when the collision point is on a flat side of the stadium the tangent coincides with that side.

Figure 2.19: The tangent $\hat{T}$ and ψ

The variable ψ is the angle measured counter-clockwise between the tangent and the post-collision velocity vector of the billiard; this relationship is shown in Figure 2.19. This gives the following range for ψ: $0 \leq \psi \leq \pi$.

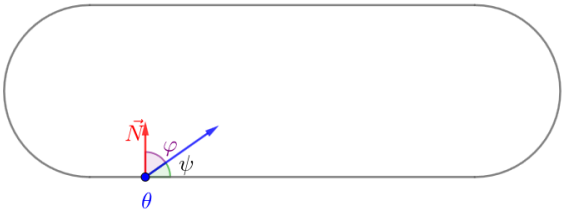

Figure 2.20: The inward normal $\hat{N}$ with φ and ψ

Traditionally, in billiard literature, $\varphi = \frac{\pi}{2} - \psi$ is used to indicate the direction of a billiard, as in Figure 2.20. The variable φ is the angle between the post-collision velocity vector and the inward normal; it is in the range $-\frac{\pi}{2} \leq \varphi \leq \frac{\pi}{2}$. s is the position on the boundary $\partial\mathcal{D}$ where the collision has occurred and is usually measured in terms of the arclength of $\partial\mathcal{D}$. Notice that this makes s a cyclic variable because it ranges from 0 to the total arclength of $\partial\mathcal{D}$ and 0 is identified with this total arclength. The situation is the same when measuring the polar angle of a circle. With this in mind, we define θ. In this paper we shall use θ to indicate the position of a collision on the boundary $\partial\mathcal{D}$ and define it with the range $0 \leq \theta \leq 2\pi$. The variable θ is the polar angle measured from the positive x-axis, with the origin being the center of the stadium. Figure 2.21 shows θ at selected values on the boundary $\partial\mathcal{D}$. By defining θ and φ, we can now describe the collision space $\mathcal{M}$ as the set of all ordered pairs of the form (θ, φ).

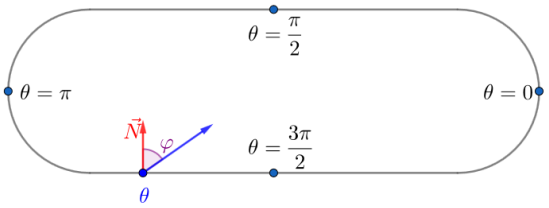

Figure 2.21: The inward normal $\hat{N}$ with φ and ψ

Since we have identified points in the collision space $\mathcal{M}$ with pairs of the form (θ, φ), we can now define distance in $\mathcal{M}$. Recall that in Sections 2.2.2 and 2.2.3 when we discussed density and transitivity we put off defining the distance between two points in the collision space $\mathcal{M}$.

Definition 2.5. *We can now define the distance function, or* ***metric****, d for the collision space $\mathcal{M}$. Let p, q be collision points where $p = (\theta_p, \varphi_p)$ and $q = (\theta_q, \varphi_q)$ then we define the* ***distance between*** *p* ***and*** *q to be* $d(p,q) = \min\left\{\sqrt{(\theta_p - \theta_q)^2 + (\varphi_p - \varphi_q)^2}, \sqrt{(\theta_p - \theta_q \pm 2\pi)^2 + (\varphi_p - \varphi_q)^2}\right\}$.

We must take the minimum of these three values because θ is cyclic. We shall discuss this further below.

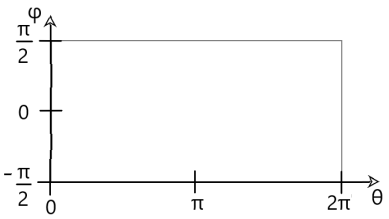

Figure 2.22: The collision space $\mathcal{M}$

If we view the collision space $\mathcal{M}$ in the plane we get the representation in Figure 2.22. Notice that for this representation to be valid the lines $\theta = 0$ and $\theta = 2\pi$ must correspond, or be identified with one another, since θ is cyclic. Such an identification can be thought of as "gluing" the lines $\theta = 0$ and $\theta = 2\pi$; this yields a cylinder, which is discussed below. We can visualize the distances measured by our metric d between two points p and q, as shown in Figure 2.23; this distance corresponds to the **euclidian distance** and is the first term inside the min function in the definition of d. The second term accounts for the occasion occurring in Figure 2.24. In Figure 2.24 the distance between p and q is actually the sum of the two blue lines because θ is cyclic. We refer to the second distance as the **wrap around distance**, the actual distance between two points is just minimum between the euclidian and wrap around distances. Since we have defined the collision space $\mathcal{M}$ and its metric d we may now discuss visual representations of $\mathcal{M}$.

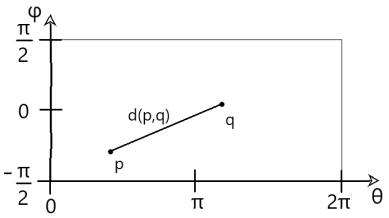

Figure 2.23: The metric d for the collision space $\mathcal{M}$

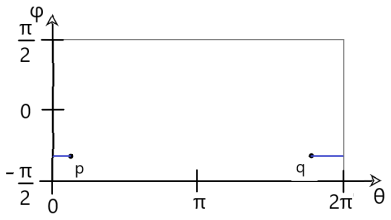

Figure 2.24: The metric d for the collision space $\mathcal{M}$ with "wrapping"

Representations of the Collision Space

In a cylindrical coordinate system a triple (r, θ', z) corresponds to a point in three-dimensional space where the **radius** r is the distance from the origin, θ' is the **polar angle**, and z is the coordinate for the **z-axis**. Now consider the points of the form $(1, \theta, \varphi)$ using the definitions of θ and φ above. This corresponds to the surface of a cylinder, omitting its flat sides, with radius one centered at the origin. Notice that the points on this cylinder are in a one-to-one correspondence with the points in the collision space $\mathcal{M}$, and since the lines $\theta = 0$ and $\theta = 2\pi$ are glued together we can view $\mathcal{M}$ as a cylinder. Formally, we say that $\mathcal{M}$ is **topologically equivalent** to a cylinder. The cylinder representation of $\mathcal{M}$ is shown in Figure 2.25.

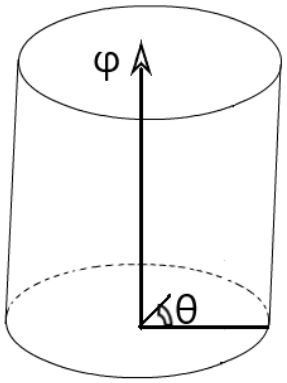

Figure 2.25: The cylinder representation of the collision space $\mathcal{M}$

As previously indicated, the goal of this work is to create a novel visualization of the evolution of the Bunimovich stadium system. The *Bunimovich Stadia Evolution Viewer*, the software which generates the visualization, uses the **cylindrical representation**. Because it is difficult to see a three dimensional object in the two dimensions of a computer screen we use the version of the cylinder depicted in Figure 2.22.

There is another way to view the collision space $\mathcal{M}$. Remember that in a spherical coordinate system a triple (r, θ', φ') corresponds to points in three dimensional space as in Figure 2.26.

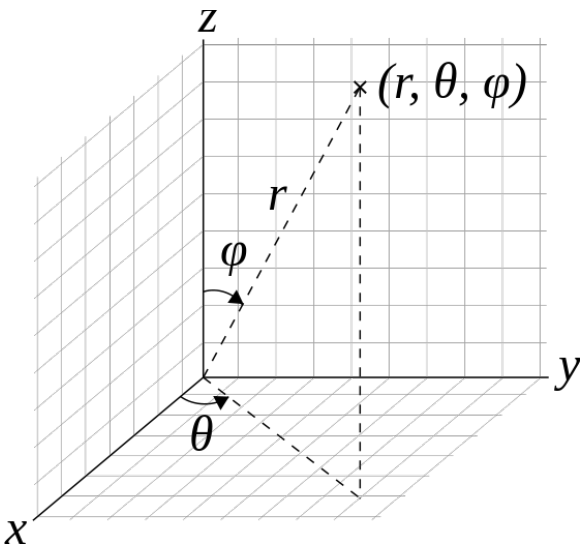

Figure 2.26: A Spherical Coordinate System. This images is courtesy of [7]

Now consider points of the form $(1, \theta, \varphi + \frac{\pi}{2})$ with θ and φ defined as usual; notice that each of these ordered triples corresponds to a point on a sphere of radius 1. In this representation θ gives the longitude and $\varphi + \frac{\pi}{2}$, which is ψ above, gives the latitude of a point on the sphere. In other words, we can think of $\mathcal{M}$ as the surface of a sphere[4]. Notice that we have allowed φ to obtain the values $-\frac{\pi}{2}$ and $\frac{\pi}{2}$ so that the poles of the sphere are included. A billiard corresponding to a point in the collision space with one of these values has a direction equal to the direction of the tangent[5] of its location on the boundary $\partial\mathcal{D}$. A billiard having a post-collision direction equal to that of its tangent is equivalent to the billiard "riding" or "skirting" the side indefinitely. Such a situation can't naturally occur in the stadium; for if a billiard had a post-collision direction of $-\frac{\pi}{2}$ or $\frac{\pi}{2}$ then the pre-collision direction would also had to have been $-\frac{\pi}{2}$ or $\frac{\pi}{2}$[6]. Therefore, this situation can only occur if a billiard's initial state specifies that $\varphi = -\frac{\pi}{2}$ or $\varphi = \frac{\pi}{2}$. We let $\mathcal{M}'$ denote the collision space with all such collision points in addition to the regular collision space $\mathcal{M}$ and call this the **spherical representation**. The reason we allow this possibility is because $\mathcal{M}'$ is topologically equivalent to the sphere (see the definition of topologically equivalent below). If we did not allow φ to take on $-\frac{\pi}{2}$ or $\frac{\pi}{2}$ then $\mathcal{M}$' is simply $\mathcal{M}$, which is topologically equivalent to a cylinder.

Definition 2.6. *When we say two spaces are "topologically equivalent" we mean two things:*

(1) There is a one-to-one correspondence between the points in the two spaces, and

(2) If two points are close in one space then their corresponding points are close in the other space. In the stadium we mean close in the sense of the metric d.

[4]This idea is due to a reader of this thesis, Bryce Weaver.
[5]Or the negative of the tangent.
[6]This is a consequence of requiring the angle of reflection to be equal to the angle of incidence.

The motivation of desiring $\mathcal{M}$ to be equivalent to the sphere is because the sphere is **compact**, meaning, it is topologically closed and bounded. **Bounded** in this context means that there is a number $M > 0$ for which the distance between any point on the sphere and the origin is less than M. In the case of the sphere corresponding to $\mathcal{M}$' we can set M to be any number greater than 1 because the sphere has radius 1. To understand what is meant by "closed" in this context recall that the collision space $\mathcal{M}$ is the surface of a cylinder where the flat sides are missing. In addition, since $-\frac{\pi}{2} < \varphi < \frac{\pi}{2}$, the cylinder does not contain values where $\varphi = -\frac{\pi}{2}$ or $\varphi = \frac{\pi}{2}$ so the "rims" of the cylinder are not included either; it is for this reason that the cylinder is not closed. Consider a φ-value near $\frac{\pi}{2}$ for a collision point in $\mathcal{M}$. Since φ must be less than $\frac{\pi}{2}$ we can always find a φ' where $\varphi < \varphi' < \frac{\pi}{2}$. If this situation can't occur then the resulting space is closed. We can prevent such a possibility by adding in the "rims". Specifically, we collapse each rim to a point and "glue" these points to the deformed cylinder. What results is topologically equivalent to a sphere. In topology this is referred to as a **two point compactification**. The motivation of transforming the cylinder representation $\mathcal{M}$ to the spherical representation $\mathcal{M}$' is that it is "easier" to deal with objects that are topologically compact. For an example of the benefits of compactness in dynamical systems, we refer the reader to [12], where compactness is used to prove a bound for the growth rate of periodic orbits in similar dynamical systems. The *Bunimovich Stadia Evolution Viewer* software produced in conjunction with this work also provides visualizations in the spherical representation.

Because we now have a better description of the collision space we define the collision map in the next section.

2.3.4 The Collision Map $\mathcal{F}$

The collision map $\mathcal{F}$ takes a position-velocity pair representing a collision and gives the next collision. In our discussion of dynamics we defined the flow Φ^t to be the function that described a billiard system's dynamics. That function gave us information about where a billiard was at a given time t. Recall that the associated phase space Ω was the set of all position velocity pairs. In our discussion of orbits we discussed how there was a one-to-one correspondence between the trajectory of a billiard, given by Φ^t, and its sequence of collisions. This meant that we only needed to consider the collision space $\mathcal{M}$ instead of Ω, thus simplifying our study of the stadium. Just as Φ^t gave us the location of a billiard given an initial condition and a time t, there is a function that gives us the next collision of a billiard given an "initial" collision. To formalize this we write $\mathcal{F} : \mathcal{M} \to \mathcal{M}$. With the representation of $\mathcal{M}$ in Figure 2.22 in mind, we can think of $\mathcal{F}$ as mapping a point in this space to another point in this space which represents the subsequent collision. In our discussion of

chaotic billiards, we referred to systems as hyperbolic when it is actually their map[7] that is hyperbolic. One may be curious if the collision map $\mathcal{F}$ is hyperbolic. For any Bunimovich billiard, including the stadium, $\mathcal{F}$ is hyperbolic when almost every trajectory is not a **type B trajectory**. In our discussion of chaotic billiards and the quantification of sensitive dependence we defined type B trajectories to be the trajectories that only collide with the flat boundary components, as in Figure 2.10. It turns out that almost every trajectory is not a type B trajectory in the stadium, so this system is hyperbolic. In this context almost every point not being type B means that the probability of randomly choosing a type B point from the collision space is zero. The probability of choosing a point from a particular subset A of $\mathcal{M}$ is the area of A divided by the area of $\mathcal{M}$. From Figure 2.22 we can see that the area of the rectangle, and thus the area of $\mathcal{M}$, is its length times its width, yielding $2\pi^2$. To find the area of the set corresponding to type B trajectories notice that these trajectories are points of the form $(t, 0)$ where t is any θ-value along a flat wall. These points are all on the line segment[8] $\varphi = 0$, and the area of this line is 0, thus the probability of selecting a type B point is 0. This means that $\mathcal{F}$ is hyperbolic.

As previously stated, the goal of this work is to visualize how nearby trajectories in the stadium diverge over time. To see how much a certain set of nearby trajectories diverge it would be useful to compute $\mathcal{F}(\theta, \varphi)$ for each ordered pair (θ, φ) in the set; this is referred to as the image of the set. To be more precise, given a subset of $\mathcal{M}$, say S, the image of S is given by $\mathcal{F}(S) = \{\mathcal{F}(x) : x \in S\}$; this is the set of all collisions occurring immediately after each collision in S. By viewing the image of a set S one could see how much divergence has occurred after one collision. To know where a collision will occur after n bounces, given a starting point $x = (\theta, \varphi)$, we compute the recurring function $\mathcal{F}^n(x)$. For example $\mathcal{F}^1(x) = \mathcal{F}(x)$, $\mathcal{F}^2(x) = \mathcal{F}(\mathcal{F}(x))$, etc. To find the image of a set $S \subset \mathcal{M}$ after n bounces we compute $\mathcal{F}^n(S) = \{\mathcal{F}^n(x) : x \in S\}$. For example, let A be the set corresponding to the ten collision points of Figure 2.27; these correspond to the line segment in Figure 2.28. $\mathcal{F}(A)$ is the set pictured in Figure 2.29. The cylinder view for the points in the set $\mathcal{F}(A)$ is shown in Figure 2.28.

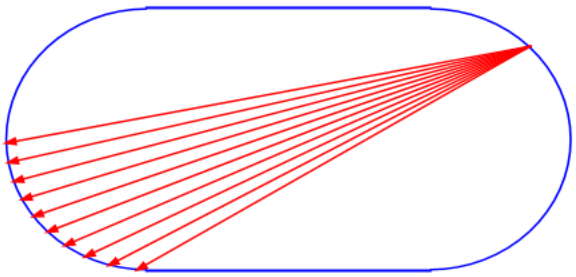

Figure 2.27: The set A of ten collision points in the stadium

[7] A system can also be established to be hyperbolic based on the flow. Here we only discuss the discrete collision map and its relation to hyperbolicity.

[8] Notice that the points in type B trajectories don't make up the entire line $\varphi = 0$.

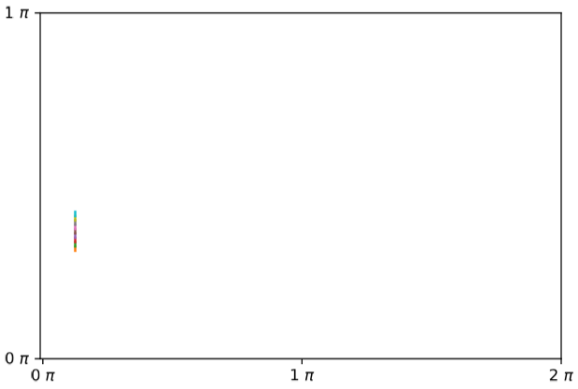

Figure 2.28: Cylinder representation of the points in A; these correspond to the points in Figure 2.27.

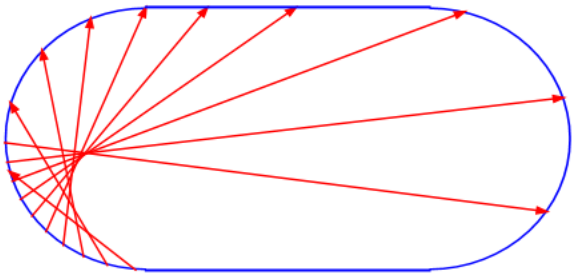

Figure 2.29: The image $\mathcal{F}(A)$ of the set of collision points in Figure 2.27

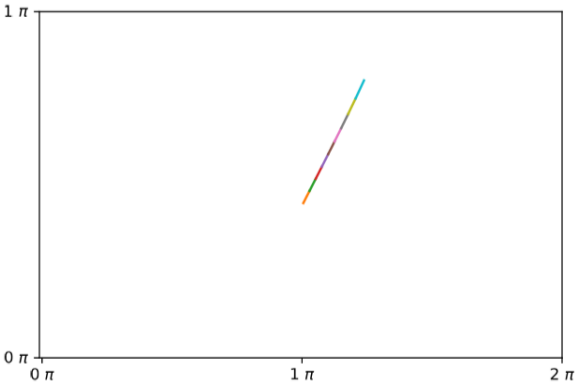

Figure 2.30: Cylinder representation of $\mathcal{F}(A)$; these correspond to the points in Figure 2.29.

We can now be more precise about our goal, which is to develop software that computes $\mathcal{F}^n$ for a given set and presents the results graphically in the spherical representation of $\mathcal{M}'$, in the cylindrical representation of $\mathcal{M}$, and in the stadium as in Figures 2.27 and 2.29. In fact, the software developed in conjunction with this project, the *Bunimovich Stadia Evolution Viewer*, was used to create the images in Figures 2.27, 2.28, 2.29, and 2.30. Notice that between Figures 2.27 and 2.29 it is difficult to tell where particular billiards end up in relation to where they started without "tracing" their trajectories; this is the motivation of providing the

cylinder representation. The cylinder representation allows one to see how billiard paths which are initially adjacent diverge over time.

In the following section we discuss a property of $\mathcal{F}$ which has important implications for the *Bunimovich Stadia Evolution Viewer*.

2.4 The Homeomorphism $\mathcal{F}$

Before we begin a discussion of homeomorphisms we must first establish the definitions of *inverse* and *continuity*. A map, $f : A \to B$, is said to be **invertable** if there exists another map $g : B \to A$ such that for every $x \in A$ we have that $g(f(x)) = x$ and for every $y \in B$ we have that $f(g(y)) = y$. If such a map g exists then we call g the **inverse** of f and denote it $f^{-1} = g$. The inverse of a map can be though of as "undoing" the map and similarly, a map can be though of as "undoing" its inverse. For instance, in the above definition when f is applied to the input x and g is applied to the result, we get x back, thus g "undoes" f. For a similar reason we can think of f as "undoing" g. The inverse function $f^{-1} : B \to A$ exists if and only if f is a bijection.

The collision map $\mathcal{F}$ for the Bunimovich stadium is invertable [5]. We can make sense of this by noting that an inverse of $\mathcal{F}$ would take a collision point (θ, φ) and find the previous collision point $\mathcal{F}^{-1}((\theta, \varphi))$. To see how this is accomplished observe the red line in the upper left plot in Figure 4.1 of Section 4.2.1. The billiard travelling along the link corresponding to this collision point leaves the bottom wall and strikes the top wall[9]. This collision point, along with every collision point, is represented by a pair (θ, φ). Recall that φ is the angle the post-collision velocity vector makes with the inward normal as described in Section 2.3.3 and is in the range $(-\frac{\pi}{2}, \frac{\pi}{2})$. Notice that if we want to find the previous collision point all we need to do is reflect φ along the inward normal, obtaining a new collision point, and then find its succeeding collision point and reflect this point's φ-value along the inward normal. For convenience we adopt the notation $-(\theta, \varphi) = (\theta, -\varphi)$. Reflecting φ along the inward normal turns (θ, φ) into $(\theta, -\varphi)$ and finding its next collision point amounts to simply computing $\mathcal{F}((\theta, -\varphi))$. Reflecting the resulting point's φ-value again can be written as $-\mathcal{F}((\theta, -\varphi))$. This yields the equality

$$\mathcal{F}^{-1}((\theta, \varphi)) = -\mathcal{F}((\theta, -\varphi)) \tag{2.1}$$

Before defining homeomorphism we need the following definition

[9] We remind the reader that *link* and *wall* are defined in Sections 2.2.2 and 2.3.1 respectively.

Definition 2.7. *We say that a map $f : A \to B$ is **continuous at a point** $c \in A$ if for every $\epsilon > 0$ there is a $\delta > 0$ such that for every x where $d(x, c) < \delta$ it follows that $d(f(x), f(c)) < \epsilon$. We say that f is **continuous** if it is continuous at every point in its domain A.*

It turns out that for any chaotic billiard, including the Bunimovich stadium, the collision map is continuous at every point that is not on a *dispersing wall* where the map is defined [5]. Dispersing walls are defined in Section 2.3.1. In general a collision map can only fail to be defined when there are *cusps*. A **cusp** occurs when two walls meet as shown in Figure 2.31; in this figure billiards may move in the non-shaded region between the circles and the enclosing rectangle. There are two cusps on either side of where the circles meet. Since there are no dispersing walls nor cusps in a Bunimovich stadium, we can conclude that $\mathcal{F}$ is continuous. Note too that since $\mathcal{F}$ is continuous $\mathcal{F}^{-1}$ is continuous too; because by Equation 2.4 $\mathcal{F}^{-1}((\theta, \varphi)) = \mathcal{F}((\theta, -\varphi))$.

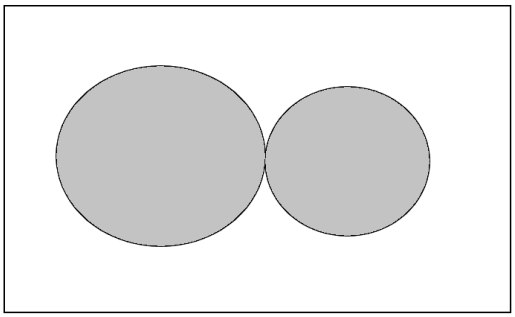

Figure 2.31: An example of a billiard with a cusp

Since we have defined continuity and the inverse of a function we can define a homeomorphism.

Definition 2.8. *A **homeomorphism** is a continuous map $f : A \to B$ with a continuous inverse.*

Since the collision map $\mathcal{F}$ is continuous and its inverse is continuous too it must be the case that $\mathcal{F}$ is a homeomorphism. It is well known that a homeomorphism has some wonderful qualities most of which are beyond the scope of this work. One such quality is that a homeomorphism maps *simple curves* without *loops* to *simple curves* without *loops*[10][1]. Examples of simple curves without loops are given in Figure 2.32 and examples of simple curves with loops are shown in Figure 2.33. In Chapter 5 the fact that the collision map is a homeomorphism will be important for the *Bunimovich Stadia Evolution Viewer* (BSEV) software. We will use this property to help ensure the correctness of visualizations produced by the software.

[10] A homeomorphism also maps simple curves with loops to simple curves with loops, even preserving the number of loops. For our purposes we are only concerned with the fact that simple curves without loops are mapped to simple curves without loops.

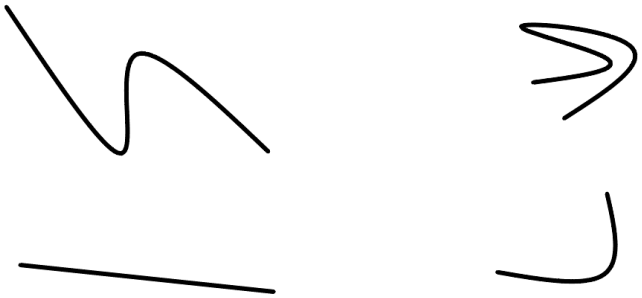

Figure 2.32: Four examples of simple curves without loops

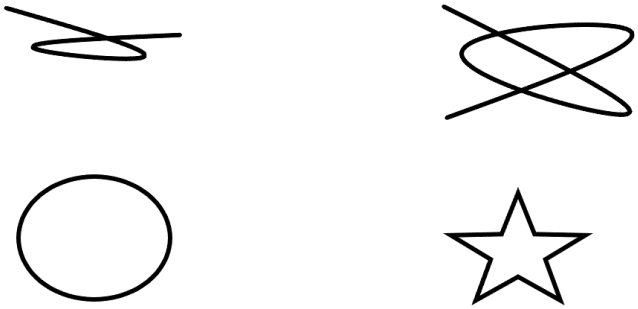

Figure 2.33: Four examples of simple curves with loops

This concludes the mathematical preliminaries necessary for understanding the BSEV and the visualizations it provides. In the following chapter we discuss current visualization tools for the Bunimovich stadium. In Chapter 4 we present the BSEV, which accomplishes the goal of this thesis. In Chapter 4 we also discuss the development of the software along with its properties. In Chapter 5 we discuss visualizations produced by the BSEV and how these can be used to gain insights into the Bunimovich stadium's chaotic behavior. We also make the case that such insights are nearly impossible to infer from the existing visualization tools. In Chapter 6 we discuss our conclusions and in Chapter 7 we discuss future venues of research with respect to this work.

Chapter 3

Review of Existing Tools

In this chapter we discuss two current visualization tools for the Bunimovich stadium. These tools were briefly mentioned in Chapter 1. As of this writing the author is only aware of these two tools for the visualization of the stadium. The first tool was published on the American Mathematical Society's blogs by the physicist John Baez and is credited to Phillipe Roux [2]. The second tool was created by the computer scientist Carlos Scheidegger [10]. Both tools show sets of billiards with close initial conditions moving in the stadium in real time.

The tool credited to Roux [2] shows a one minute video of the billiards shown in Figure 3.1. Included in the Figures are images of the billiards at various in points time. With this tool one can see that the billiards become evenly distributed within the stadium rather quickly. One drawback of this viewer is that it only allows one to see the motion of a single set of trajectories. Scheidegger's tool [10] does not have this shortcoming.

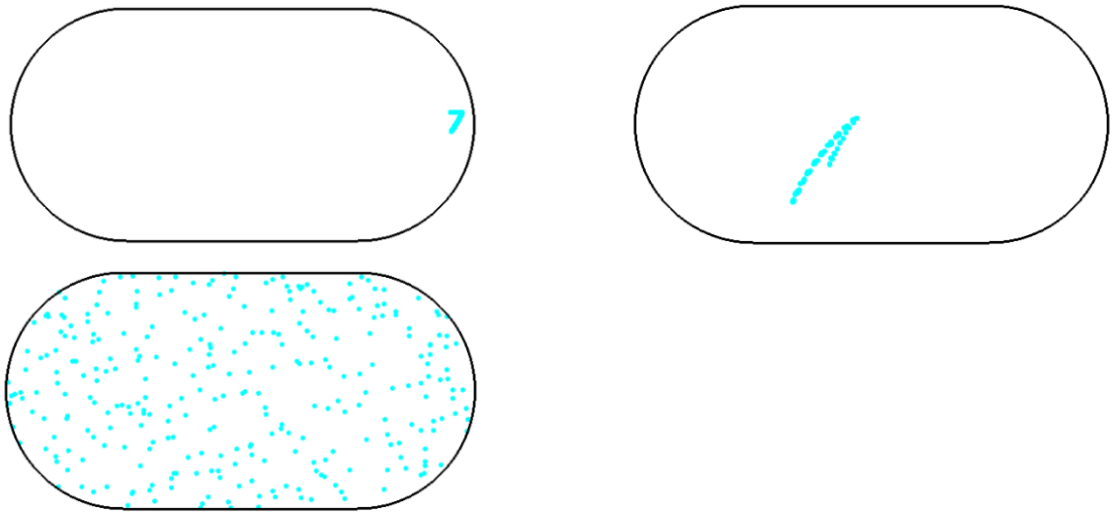

Figure 3.1: Image from the viewer on the American Mathematical Societies' blog

Scheidegger's tool gives users a choice between viewing the trajectory of one billiard or the trajectories of three hundred billiards. If one billiard is selected then a billiard starts in the middle of the stadium with a random direction and bounces indefinitely. Each time the billiard encounters the boundary a line is drawn tracing its path from the previous collision. If the option for three hundred billiards, is selected then three hundred billiards with close initial conditions start moving in the stadium. In this case lines are drawn in the same manner as with the single billiard. Figure 3.2 shows images produced by this Scheidegger's viewer with three hundred billiards. This viewer is more flexible than the one produced by Roux because it allows more user choices. Scheidegger's viewer traces the path of billiards too, this allows one to see how the trajectories diverge. After several bounces, however, it is hard to distinguish which billiards were initially closer and which were not as shown in the last four stadia in Figure 3.2. Since this viewer allows user input it is less restrictive than Roux's. However, it is still restrictive since it selects the initial conditions of billiards instead of allowing the user to do so.

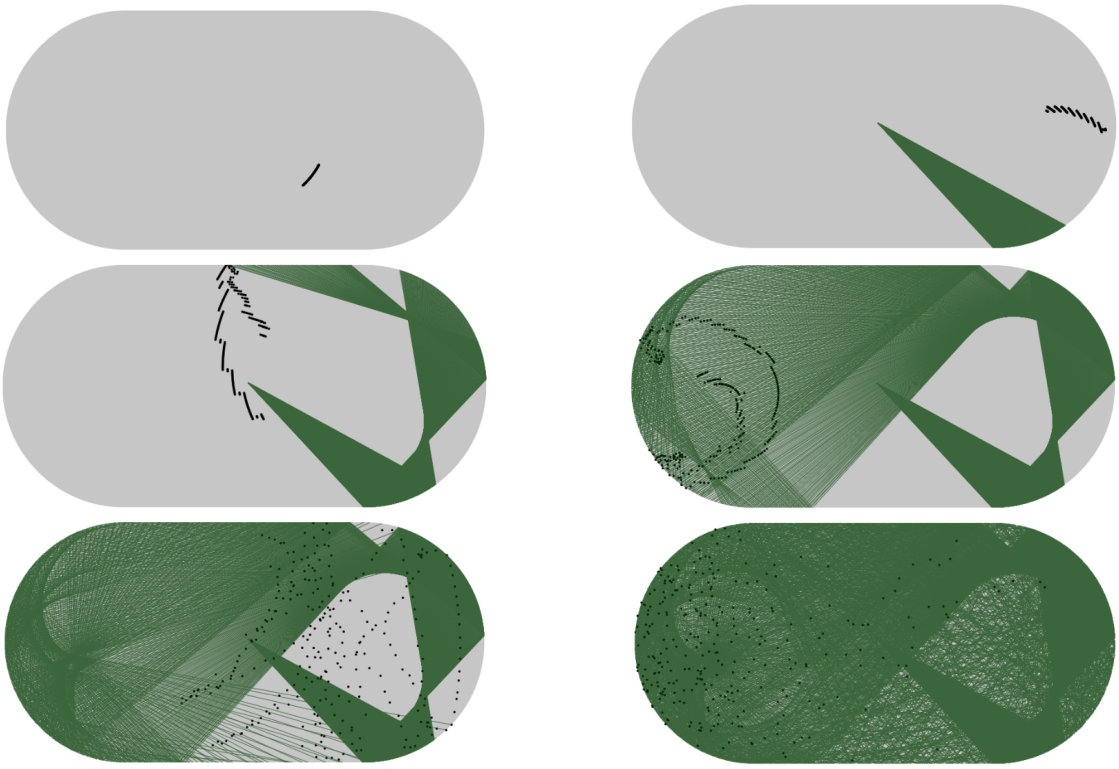

Figure 3.2: Image from Carlos Scheidegger's viewer

The *Bunimovich Stadia Evolution Viewer* is designed so that it does not have the shortcomings of Roux's and Scheidegger's tools. It allows the user to input a set of close billiard trajectories. While it does not show the billiards moving in real time it does allow for the visualization of divergence as we discuss in Chapter 5.

In the following chapter we discuss the design an implementation of the BSEV. In Chapter 5 we discuss the software's strengths and argue that insights gained using the BSEV are nearly impossible to make using current viewers.

Chapter 4

The Design and Implementation of the *Bunimovich Stadia Evolution Viewer*

In this chapter we present the *Bunimovich Stadia Evolution Viewer* (BSEV) software, which computes images of close trajectories under iteration of the collision map $\mathcal{F}(x)$, as discussed in Section 2.3.4. We discuss the objectives for the software and the development process used to meet these objectives. We also describe the implementation details of the BSEV. The software can be freely downloaded via Github at the following URL: https://github.com/shoemarw/BunimovichStadium. Its code is included in Appendix A.

4.1 Objectives for the BSEV

The goal of this thesis, as specified in Chapter 1 and more precisely in Chapter 2, is to produce software that computes $\mathcal{F}^n(A)$ for some set of close trajectories A and present a *novel* visualization of the resulting set. The BSEV software which achieves this goal must have the ability to carry out the following tasks.

1. It must be able to compute $\mathcal{F}(x)$ for a given collision point x.
2. It must be able to plot $\mathcal{F}(x)$ in the stadium view.
3. It must be able to compute $\mathcal{F}^n(x)$ for a given collision point x and a natural number n. This can only be achieved after 1 is achieved.
4. It must be able to compute $\mathcal{F}(A)$ for some set A of close collision points.

5. It must be able to compute $\mathcal{F}^n(A)$ for some natural number n and set of collision points A. This can only be achieved after 4 is achieved.

6. It must provide a means of sampling points from an input set A.

7. It must be able to plot $\mathcal{F}(A)$ in the stadium view as shown in as shown in Figure 1.2.

8. It must be able to plot $\mathcal{F}(A)$ in the cylindrical and spherical views as discussed in Chapter 2.

9. It must provide a means for the user to supply the inputs necessary for its function.

The BSEV achieves each of the above objectives. The BSEV was developed in four stages. The first stage produced a prototype that achieved objectives 1, 2, and 3. The second stage produced a prototype that achieved objectives 4, 5, 6, and 7 in addition to those achieved by the first prototype. The third prototype achieved the same objectives as the second in addition to objective 8. The prototype produced by the fourth and final stage achieved all of the objectives. In the following section we discuss the development of the BSEV.

4.2 Developement of the *Bunimovich Stadia Evolution Viewer*

The programming language Python version 2.7.15 was used to create the BSEV software. There are two reasons for this choice: the ability to use matplotlib and the author's desire to gain more experience in using Python. Matplotlib is a software library which enables the plotting of geometric figures; matplotlib was used to generate all images created by the BSEV software. The software was developed using an agile development model; this was so there would always be a working prototype during every stage of the development process. The prototypes were useful in making sure that every component of the software created during a stage was functioning properly and achieving its objectives. The prototype created after each stage of the development process had the ability to produce images. This enabled the developer to ensure that the software was producing the expected output. As mentioned before, there were four stages of the development process; each produced a working prototype as described below.

4.2.1 First BSEV Prototype

The first prototype was able to produce images of single trajectories in the stadium as in Figure 4.1.

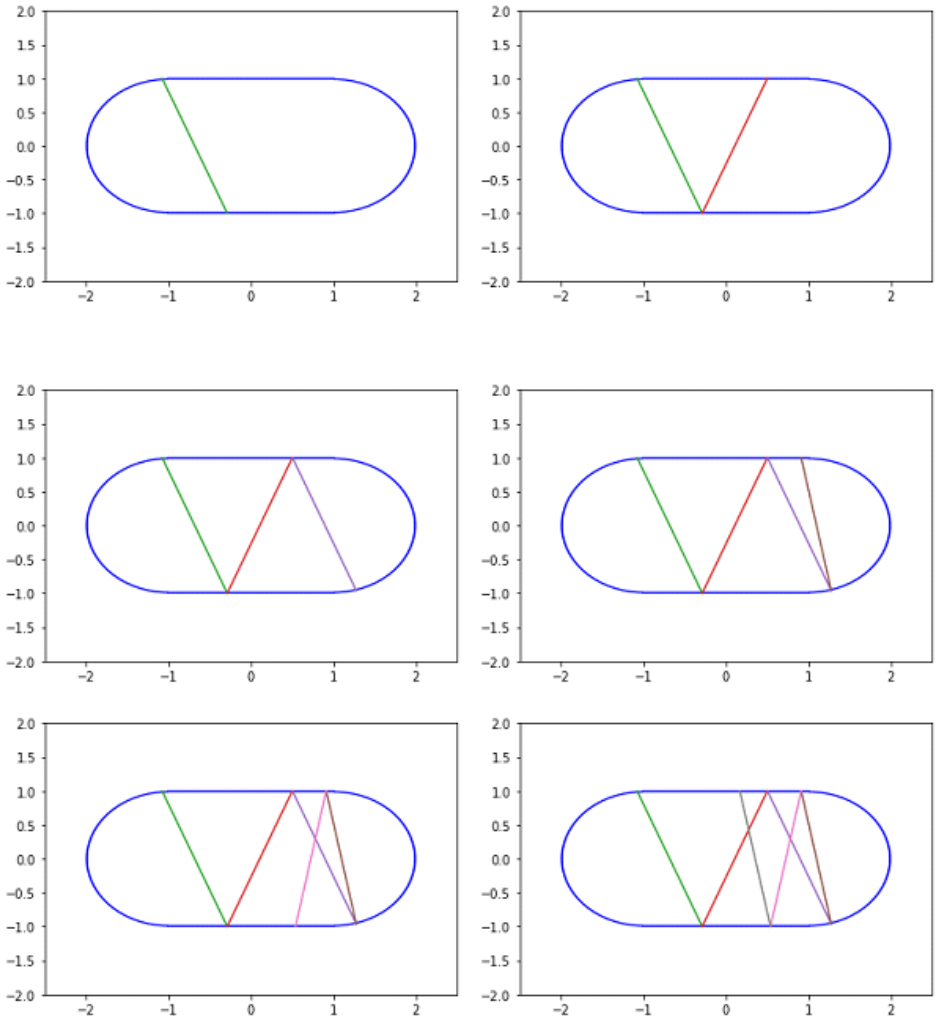

Figure 4.1: Output from the first prototype of the BSEV software; shows 5 iterations of the collision map $\mathcal{F}$ to a point

The first plot in Figure 4.1 shows a collision point, a starting at the top side and aiming towards the bottom side. The next five plots show $\mathcal{F}(a)$, $\mathcal{F}^2(a)$, $\mathcal{F}^3(a)$, $\mathcal{F}^4(a)$, $\mathcal{F}^5(a)$ respectively. Figure 4.2 shows the first 1000 collisions of the point a. This shows that the first prototype achieved objectives 1, 2, and 3 listed at the beginning of this chapter.

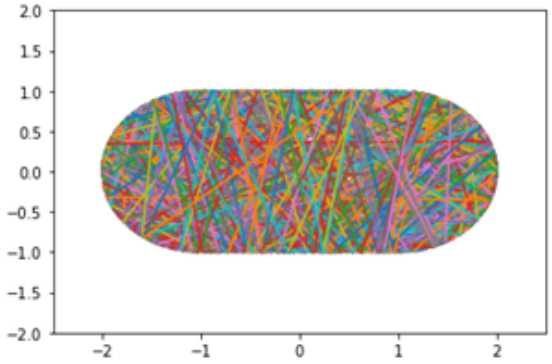

Figure 4.2: 1000 collisions of the point in Figure 4.1

In Figure 4.3 the architecture of the final version of the *Bunimovich Stadia Evolution Viewer* is depicted. It shows the "`.py`" files comprising the final version of the BSEV and the relationships between them. Development of the first prototype produced `CollisionMap.py`, an early version of `CoordinateConversion.py`, and a file not used in the final version which allowed for the plotting of images such as those in Figure 4.1 and 4.2 using the aforementioned `.py` files. The file that did the plotting for this prototype achieved objective 2. `CollisionMap.py` implements the collision map $\mathcal{F}$; its code can be found in appendix A.4. This file defines functions which work together to compute the trajectory of a given collision point. Given an initial collision point and a number of iterations, the function `collisionLoop` produces the point's trajectory by successively calling the function `collisionMap`. The function `collisionMap` takes a point and produces the next collision point; this is accomplished by running collision detectors to see which wall of the stadium is hit next. After running collision detectors, the position of the next collision is computed using the formula

$$\vec{v}^{+} = \vec{v}^{-} - 2(\vec{v}^{-} \cdot \vec{n})\vec{n}$$

where $\vec{v}^{+}$ is the post-collision velocity vector, $\vec{v}^{-}$ is the pre-collision velocity vector and $\vec{n}$ is the inward normal as discussed in Section 2.3.3. For specifics on how the formula is implemented we refer the reader to the code in Apendix A.4. The functions `collisionMap` and `collisionLoop` achieve objectives 1 and 3 respectively. The early version of `CoordinateConversion.py` implemented in the first prototype and its code can be found in Appendix A.5. This version of `CoordinateConversion.py` only contained the function `mod2pi` which takes an angle and gives the angle co-terminal to it in the range $(0, 2\pi)$.

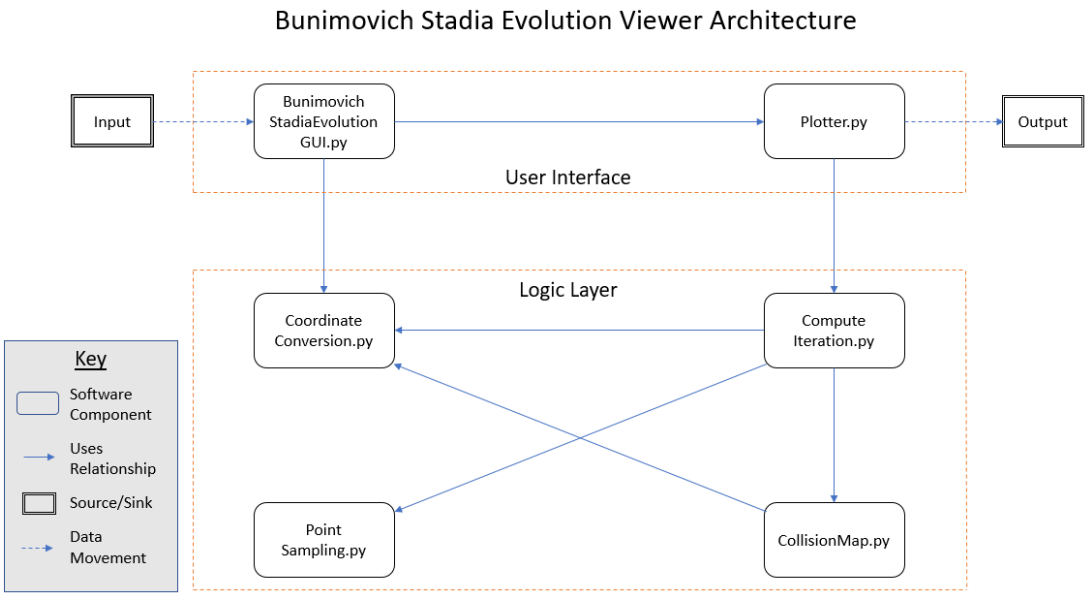

Figure 4.3: Architecture of the Bunimovich Stadia Evolution Viewer Software.

4.2.2 Second BSEV Prototype

Development of the second prototype of the *Bunimovich Stadia Evolution Viewer* produced `ComputeIteration.py`, `PointSampling.py` and the final version of `CoordinateConversion.py`. `ComputeIteration.py` can be found in Appendix A.3. This file defines two functions each of which takes a set of points and computes the trajectories of the points within. Both of these functions call the functions in `CollisionMap.py` to compute the trajectory of individual points. The first function, `image_const_theta`, takes a set of collision points with a constant θ-value and ranging φ-values and computes this set's trajectory given a specified number of iterations. The second function, `image_const_phi`, takes a set of collision points with a constant φ-value and ranging θ-values and computes this set's trajectory given a specified number of iterations. `image_const_theta` and `image_const_phi` both achieve objectives 4, and 5 from the beginning of this chapter. For these two functions the input set is specified in the same manner so we only discuss how the input set is specified in `image_const_theta`. To specify the input set for `image_const_theta` the constant value for θ along with the upper and lower bounds for the range of φ-values must be given. Such a set corresponds to a vertical line when plotted in the collision space as shown in Figure 2.28 of Section 2.3.4. Notice that a line contains an infinite number of points. Because a computer can only compute a finite number of things a finite number of samples are taken from the φ-value range. Therefore, another input to the functions `image_const_theta` and `image_const_phi` is required to specify a sampling method.

The second prototype and the subsequent versions of the BSEV provides two sampling methods. These methods are enabled in `PointSampling.py`, as shown in Appendix A.6 the functions in this file achieve objec-

tive 6 from the beginning of this chapter. The first sampling method is implemented in `evenSpacingSample`. In this method a specified number of points evenly spaced in the specified range are selected. The second sampling method is implemented in `randomSample`. In this method a specified number of points are randomly chosen according to the uniform distribution.

The second prototype also produced a final version of `CoordinateConversion.py`. The functions added during the second iteration of development convert (θ, φ) pairs to values used in `CollisionMap.py`. The second prototype plotted the trajectories of input sets as shown in Figures 2.27 and 2.29 of Section 2.3.4. Therefore, the second prototype also achieved objective 7.

4.2.3 Third BSEV Prototype

Development of the third prototype produced `Plotter.py` as seen in Appendix A.2. This file provides a function, `plotter`, which produces the desired cylinder and spherical views as well as showing trajectories in the stadium itself. Therefore `plotter` achieves objective 8. The outputs of this function are `.pdf` files of the specified plot types. The function `plotter` takes inputs specifying the input set, the sampling method used, the number of iterations to be computed and the type of plot to produce in terms of cylinder, sphere, or stadium. Examples of stadium plot outputs of this prototype are shown in Figure 4.4.

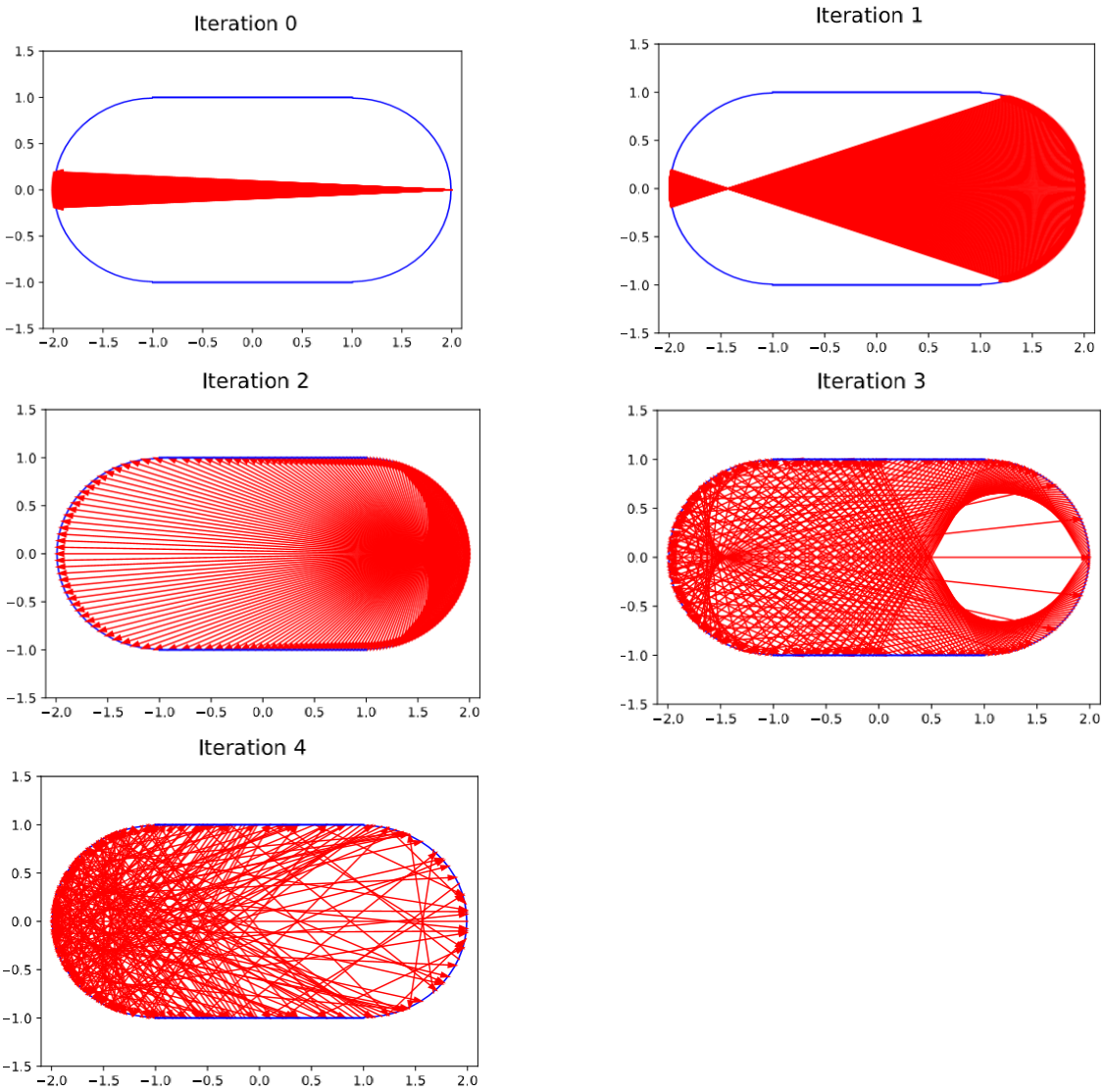

Figure 4.4: Example outputs of the third prototype of the BSEV software

Examples of cylinder plot outputs of this prototype are shown in Figure 4.5 and correspond to the plots in Figure 4.4. Examples of spherical plot outputs of this prototype are shown in Figure 4.6 and correspond to the plots of Figure 4.4.

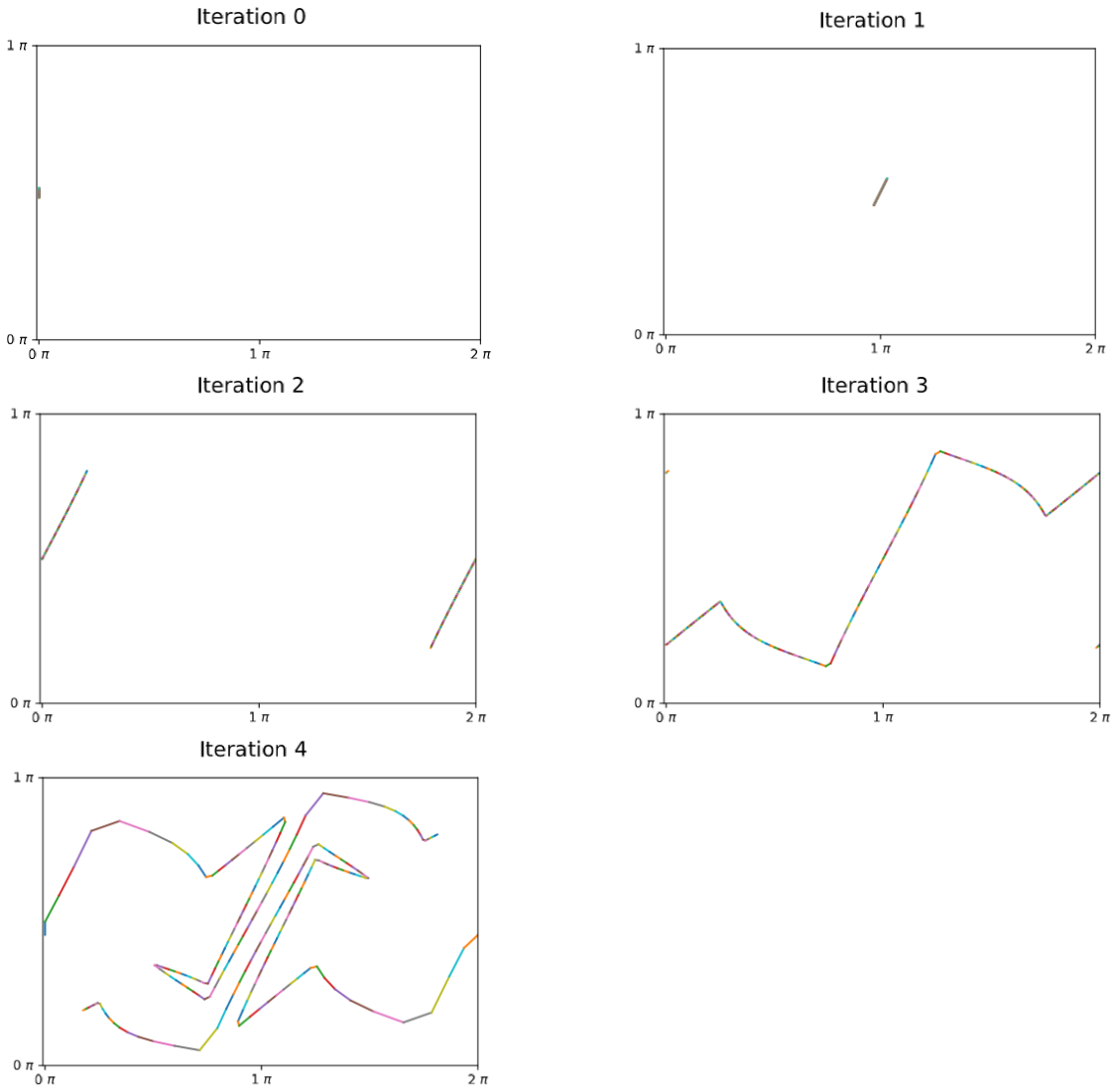

Figure 4.5: Example outputs of the third prototype of the BSEV software in cylindrical view

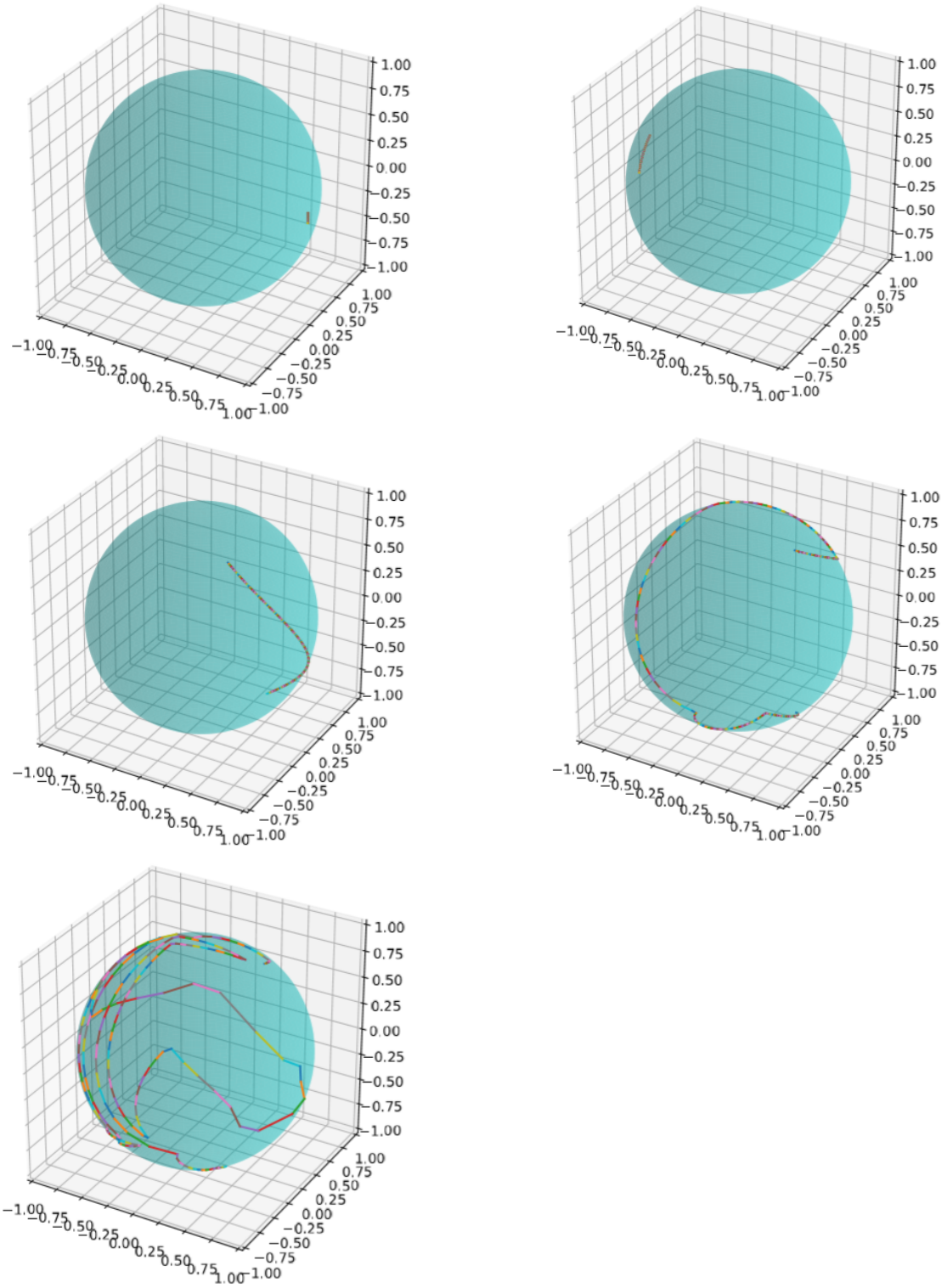

Figure 4.6: Example outputs of the third prototype of the BSEV software in spherical view

The plots of Figure 4.4 highlight the need for the cylinder and spherical plots. In these plots it is

difficult to see the divergence of trajectories over time because in later iterations it is difficult to see which collision points were close initially. In fact, when viewing the plots after three iterations of the computer representation of the collision map $\mathcal{F}$ one can't easily tell which trajectories were closer to one another initially and which ones were not. In the cylindrical and spherical plots it is easier to see which collision points were close initially and which weren't because they keep their relationship with one another; i.e. if a collision point is connected to another with a line segment initially in the cylinder or spherical plots then this will be the case after applying the computer representation of $\mathcal{F}$. The images shown in the cylindrical view illustrate the novel nature of visualizations provided by the BSEV. The evolution of the Bunimovich stadium system can be seen in an entirely new way when compared to traditional views of billiards moving within the boundary in real time.

4.2.4 Fourth BSEV Prototype and Final Product

The fourth and final stage of the agile development process produced `BunimovichStadiaEvolutionGUI.py` as shown in Appendix A.1. This file enables the graphical user interface (GUI) of the BSEV software. It takes input from the user and uses the function `plotter` from `Plotter.py` to produce `.pdf` files containing the desired plots. Therefore, the final version of the BSEV achieves objective 9 in addition all others. Instructions for interacting with the GUI and thus using the BSEV software are included in Appendix B.1. In next chapter we discuss the novel visualizations produced by the BSEV and what can be inferred from its plots. We also discuss how these visualizations can give insights into the stadium's chaotic behavior and how these insights are almost impossible to make using traditional viewers.

Chapter 5

Using the *Bunimovich Stadia Evolution Viewer*

In this chapter we discuss the results of this work and what can be inferred from plots produced by the Bunimovich Stadia Evolution Viewer (BSEV) software. We first discuss important properties of the collision map $\mathcal{F}$ that imply interesting behavior in plots produced by the BSEV software and how to modify the input, in terms of increasing the number of samples, to prevent erroneous outputs in the cylinder and spherical plots. We then discuss how the software can be used to see where the Bunimovich stadium is more or less sensitive to initial conditions.

In Section 2.4 we discussed that the collision map $\mathcal{F}$ is a homeomorphism. The fact that $\mathcal{F}$ is a homeomorphism has implications for the BSEV software. Notice that sets where one variable is constant and the other varies correspond to lines in the cylinder representation. Because the input sets to the software are lines and $\mathcal{F}$ is a homeomorphism the image of every input set is a simple curve without loops. We also have that if any set, at a given iteration of the collision map, is a simple curve without loops then its image must be a simple curve without loops. We can conclude by induction that every output of $\mathcal{F}$ must be a simple curve without loops. The previous discussion indicates that the BSEV software *should* never produce loops or intersections in the plots it produces. This implies that if there are loops or intersections in the output then the BSEV software is not consistent with the collision map $\mathcal{F}$; with this in mind, consider the following plots produced by the BSEV software. Figures 5.1 and 5.2 show the trajectory of 20 collision points for 8 iterations of $\mathcal{F}$; Figures 5.3 and 5.4 are the corresponding collision points in the cylinder representation.

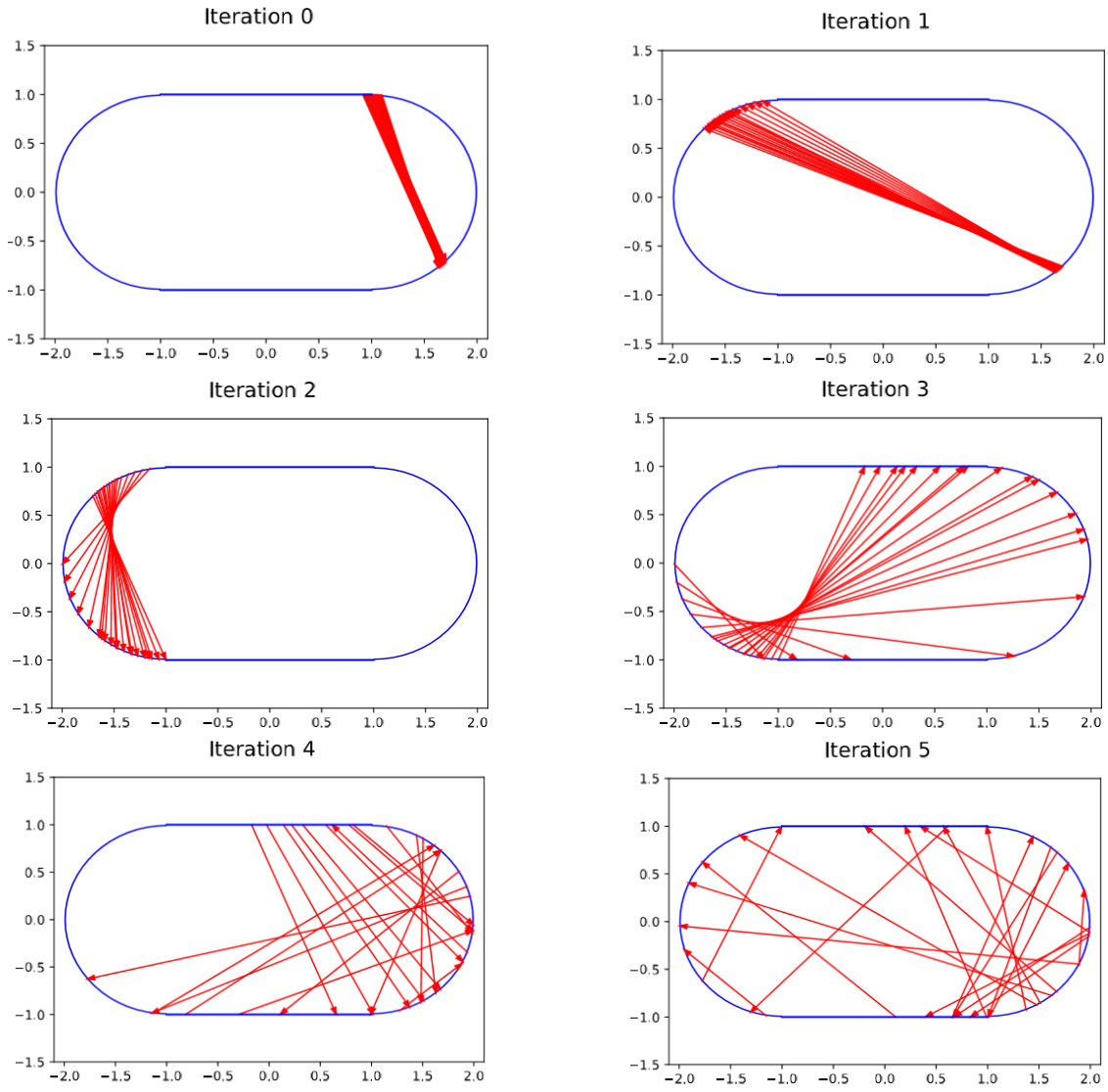

Figure 5.1: A portion of the trajectory of 20 collision points in the stadium up to iteration 5

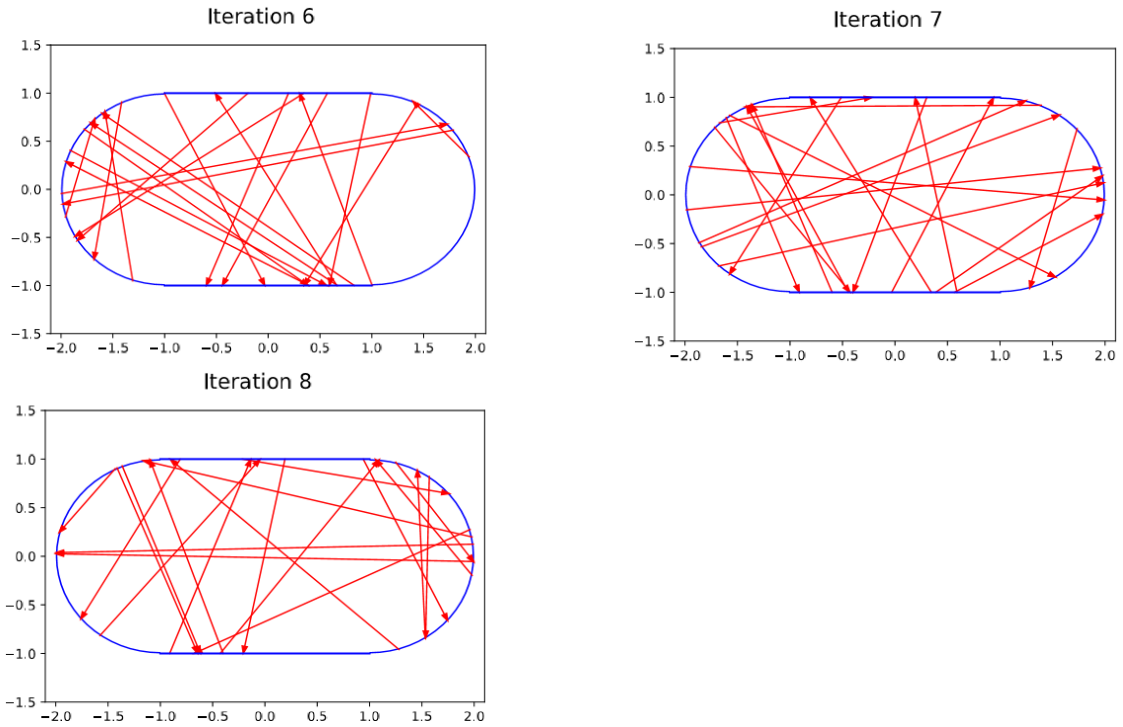

Figure 5.2: A portion of the trajectory of 20 collision points in the stadium (iterations 6 through 8)

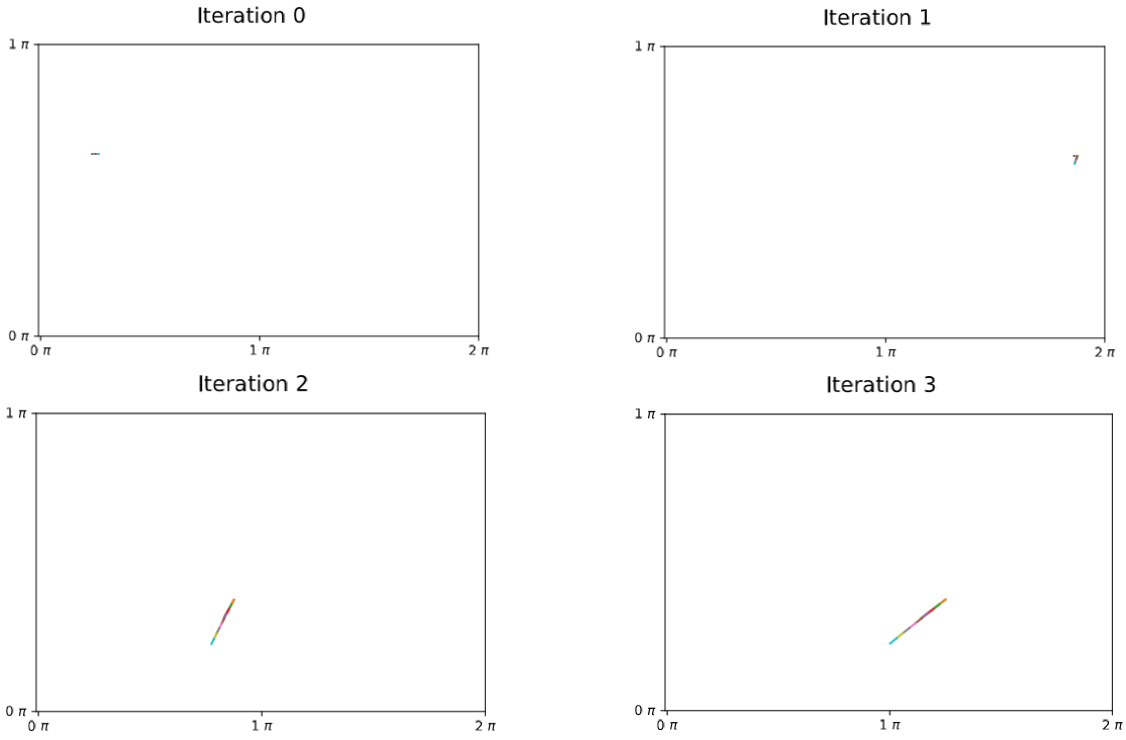

Figure 5.3: A portion of the trajectory of 20 collision points in the cylinder view for the first 3 iterations

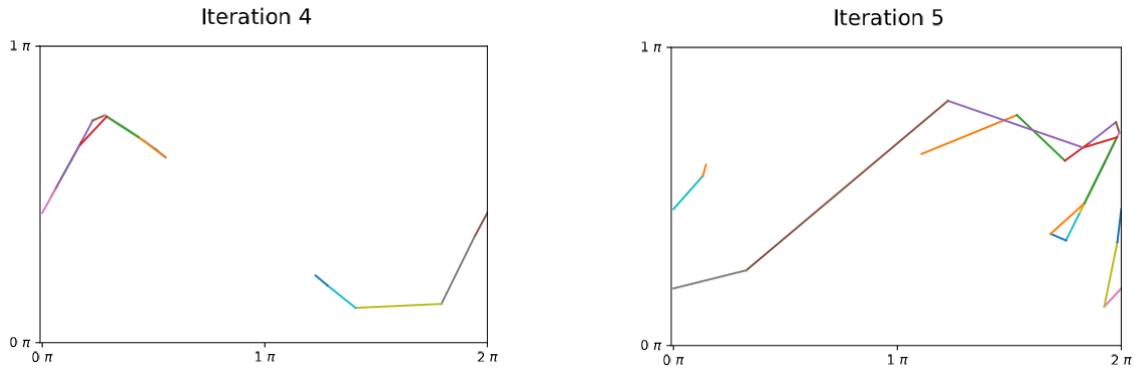

Figure 5.4: A portion of the trajectory of 20 collision points in the stadium (iterations 4 and 5)

As expected, the plots in Figure 5.3 contain no intersections. However, we can clearly see intersections in Figure 5.4. Does this mean the model is incorrect? Not necessarily. This does not mean that the BSEV software works improperly, but it does mean that these specific outputs are not accurate. To understand what caused the intersections, we must discuss **sampling**. In Section 4.2.2, we discussed that because a computer can only compute a finite number of things, the BSEV software samples points from the user provided range. We also discussed that one input provided to the software is the number of points to be sampled. In Figures 5.1, 5.2, 5.3, and 5.4 only 20 points were sampled from the input range. In order to "fix" the issue of intersections in the cylinder plots the number of sampled points must be increased. Figures 5.5 and 5.6 show cylindrical plots for the same inputs as Figures 5.1, 5.2, 5.3, and 5.4 except 200 samples are used instead of 20 and an extra iteration of the collision map is included.

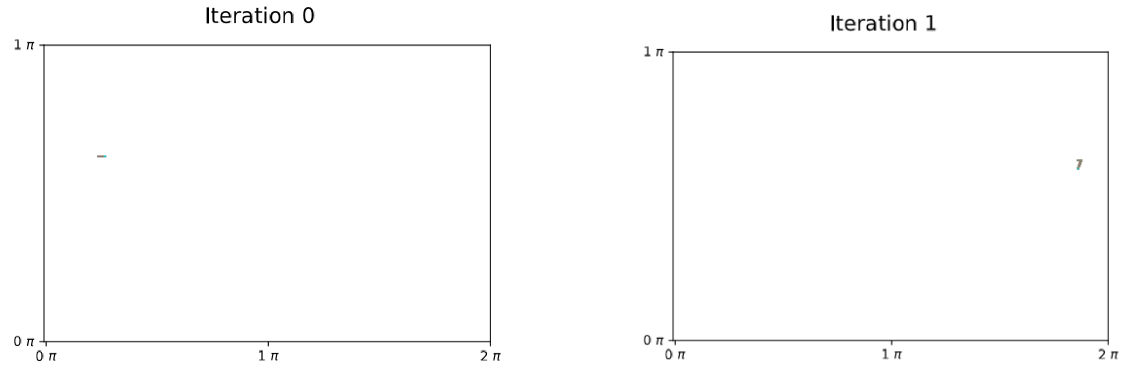

Figure 5.5: This corresponds to the motion of the same set of billiards as Figures 5.1, 5.2, 5.3, and 5.4 except 200 points were sampled instead of 20 (first 2 iterations)

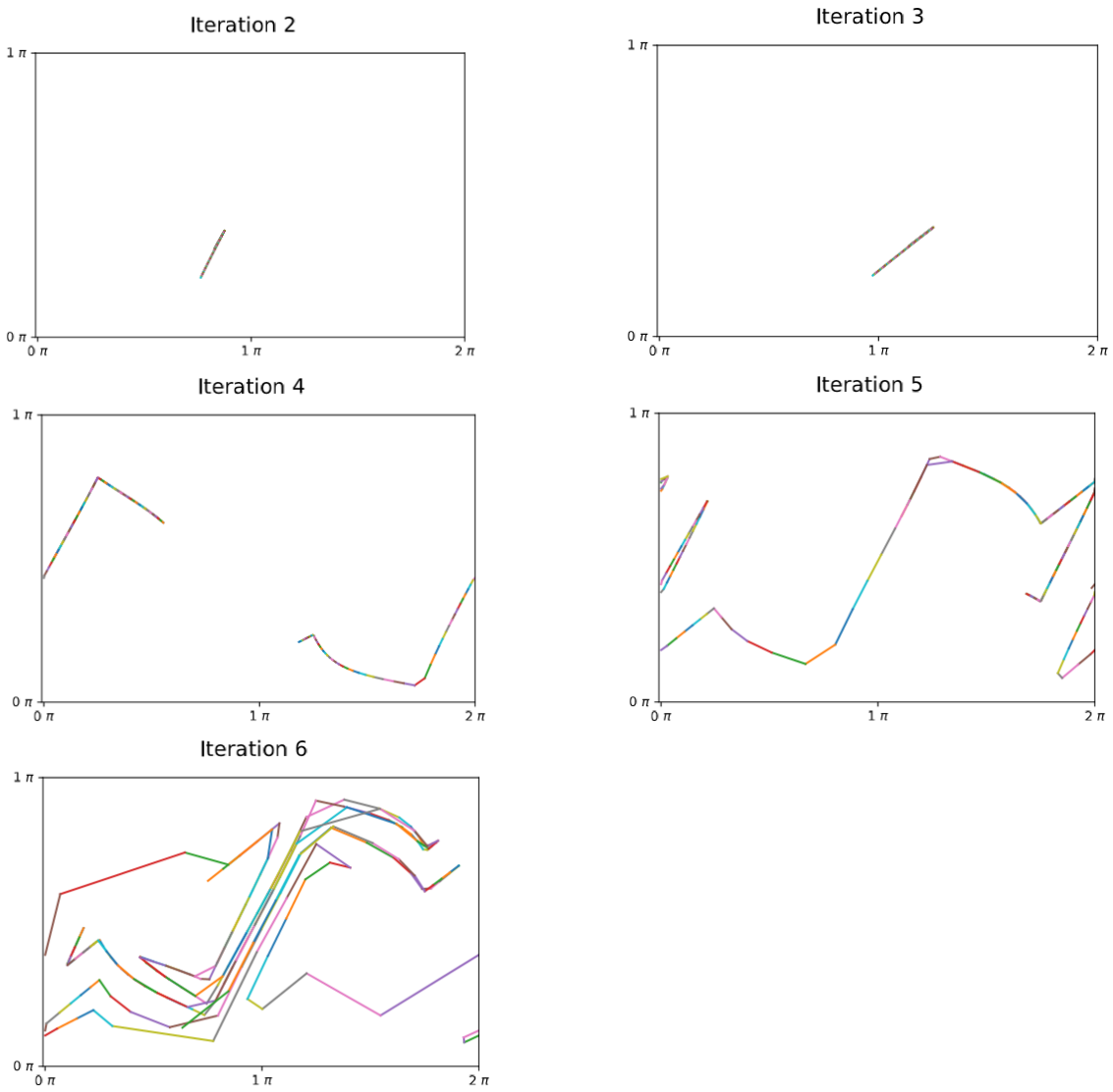

Figure 5.6: This corresponds to the motion of the same set of billiards as Figures 5.1, 5.2, 5.3, and 5.4 except 200 points were sampled instead of 20 (iterations 2 through 6)

We can see from Figures 5.5 and 5.6 that more detail has been added with respect to Figures 5.3 and 5.4 by increasing the number of samples; we sometimes refer to this as increasing the resolution. Also, the intersections in iteration 4 have been resolved and there appears to be fewer intersections in iteration 5. For iteration 6 of Figure 5.6 there are numerous intersections. By increasing the number of sampled points to 2000 the intersections in iterations 5 and 6 can be resolved as shown in Figures 5.7 and 5.8. Figure 5.8 also contains iterations 7 and 8; in the former there is only minor overlapping whereas in the latter there is significant overlap. These overlaps can be resolved too by increasing the number of sampled points. In Figure 5.9 a sample size of $10,000$ was used; here we only include iterations beyond 5 because the resolution was already good enough in previous iterations with a sample size of 2000. In iterations 7 and 8 in Figure 5.9,

there is only very minor overlap when compared to a sample size of $2,000$. Iteration 9 is included in Figure 5.9 so the reader can see that as the number of iterations is increased, no matter how large the sample size, overlap will eventually occur. The only way to guarantee no overlap is to "sample" infinitely many points, but this is not practical. In Section 5.2 we will discuss ways to maximize the number of iterations without overlap by modifying the input.

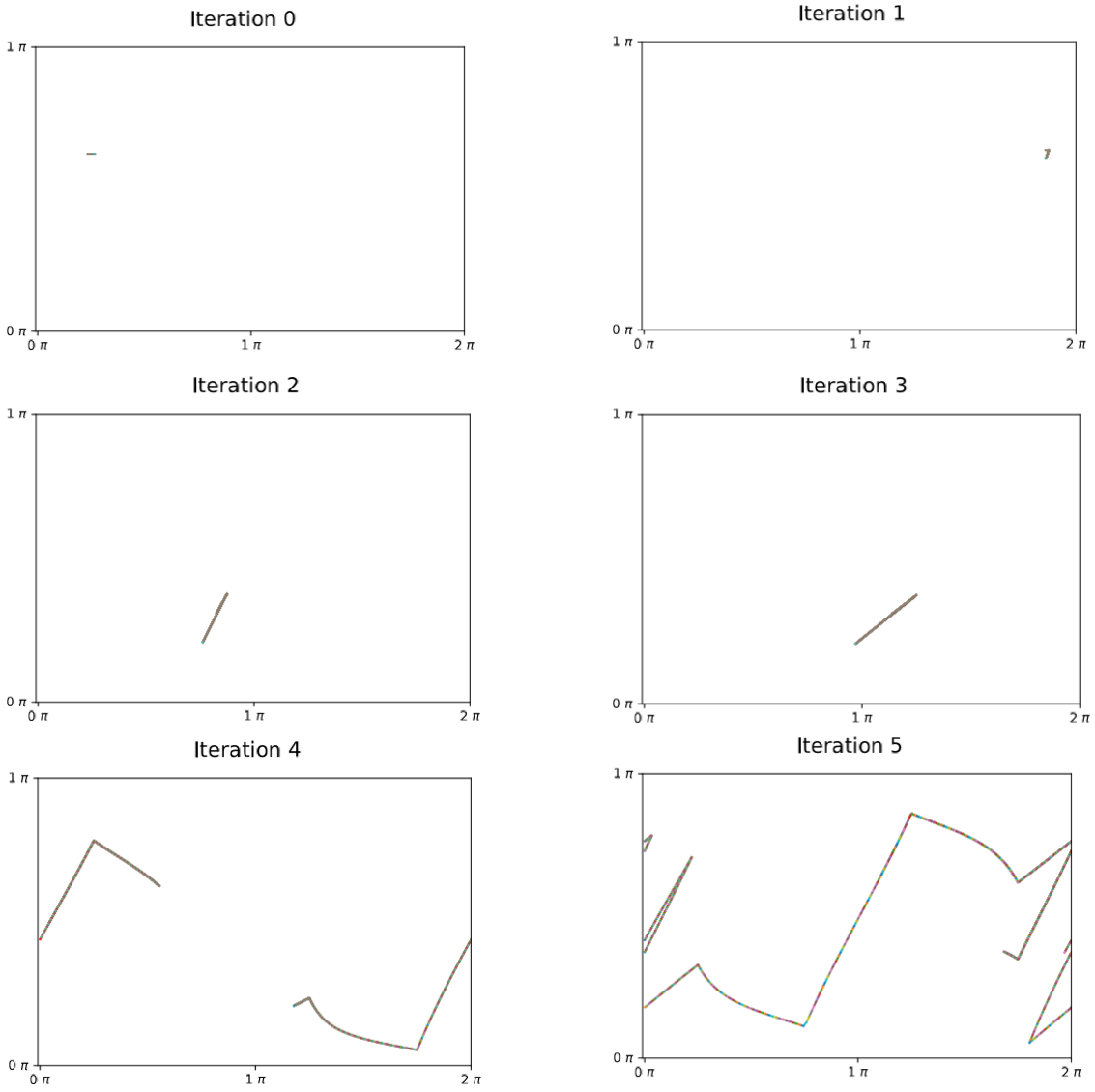

Figure 5.7: This corresponds to the motion of the same set of billiards as Figures 5.1, 5.2, 5.3, and 5.4 except 2000 points were sampled instead of 20 (up to iterations 5)

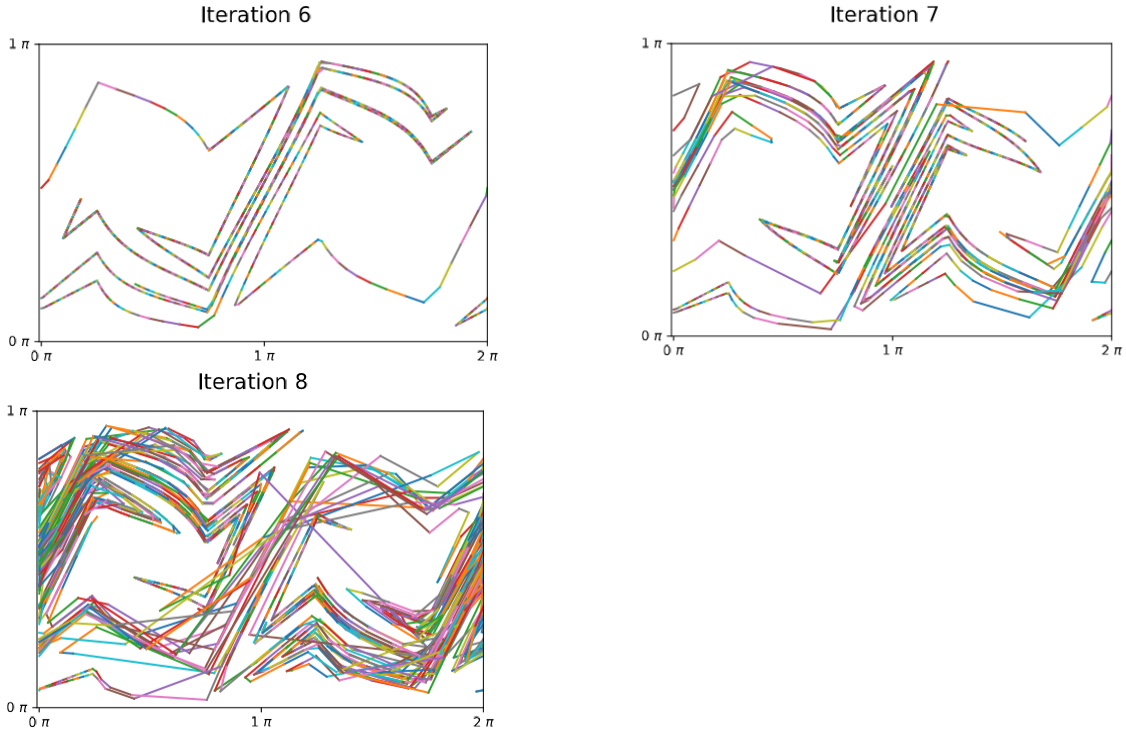

Figure 5.8: This corresponds to the motion of the same set of billiards as Figures 5.1, 5.2, 5.3, and 5.4 except 2000 points were sampled instead of 20 (iterations 6 through 8)

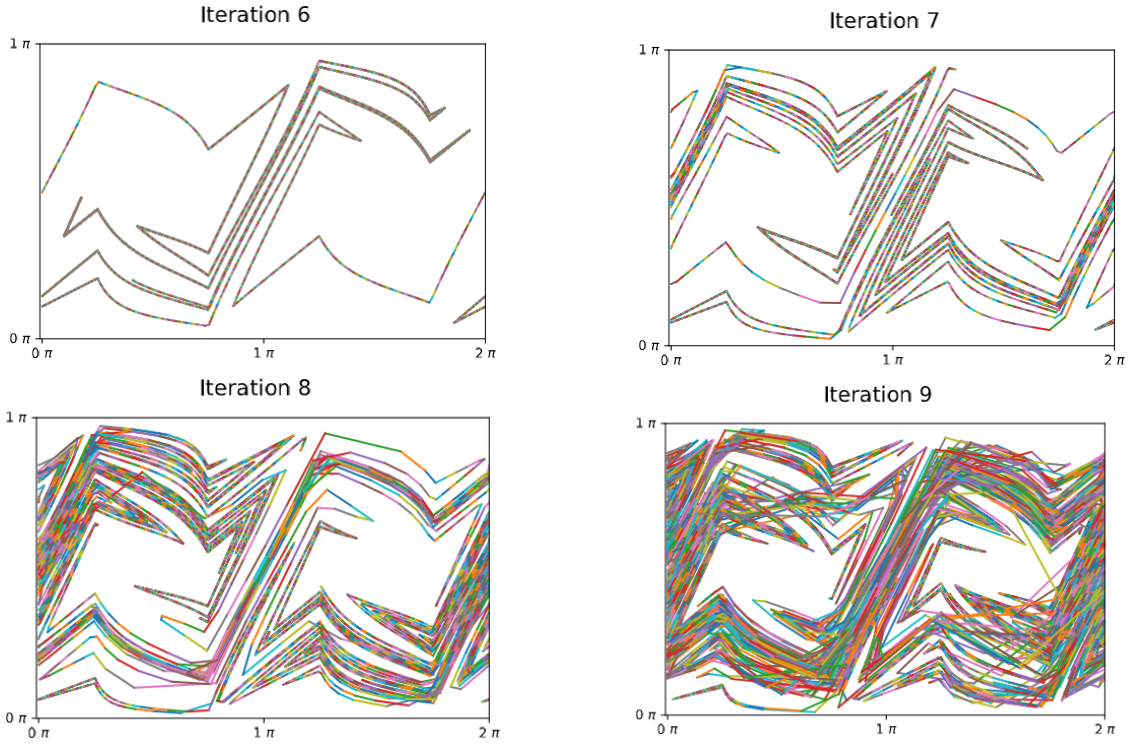

Figure 5.9: This corresponds to the motion of the same set of billiards as Figures 5.1, 5.2, 5.3, and 5.4 except $10,000$ points were sampled instead of 20 (only iterations $6, 7, 8$, and 9 are shown)

It should be noted that increasing the sample size in spherical plots has the same effect of limiting/eliminating looping, or overlap. We have included images of cylindrical plots instead of spherical because they more easily illustrate the point. In the next section we shall discuss the implications of a specific type of chaotic behavior for the *Bunimovich Stadia Evolution Viewer*.

5.1 The *Bunimovich Stadia Evolution Viewer* and Chaos

In this section we discuss the implications of an interesting property of the Bunimovich stadium that Lenoid Bunimovich proved in [4]. In his paper, "On the ergodic properties of nowhere dispersing billiards", Bunimovich showed that the stadium billiard is ergodic. The precise definition of an ergodic system is very technical and relies on many concepts not discussed in this work, these concepts are discussed in [8] and are beyond this work's scope. For the purposes of this work we shall think of the fact that the stadium is ergodic in the following way: as the number of collisions of a randomly selected point in the collision space p tends towards infinity the orbit of p will approach containing approximately every (θ, φ)-pair [2]. This implies that if a subset X of the collision space $\mathcal{M}$ is randomly selected then $\mathcal{F}^n(X)$ will approach containing approximately every (θ, φ)-pair as n tends towards infinity. With the cylinder, and spherical representations in mind, this means that as the number of iterations of the collision map tends to infinity, the plots will tend towards covering the entire cylinder (or sphere). Recall from the previous section that $\mathcal{F}$ is a homeomorphism, so if the input set is a simple curve without loops or intersections then every iteration of $\mathcal{F}$ will[1] be a simple curve without loops or intersections. This, along with ergodicity, implies that a randomly selected simple curve without loops or intersections in the collision space will approach covering the entire cylinder (or sphere) as the number of iterations approaches infinity, furthermore none of the plots of these iterations will have intersections or loops Thus it will appear that the cylinder (or sphere) has been "wrapped" by a thread in intricate patterns, such as those seen in iteration 7 of Figure 5.9, that never crosses itself; the author refers to this theoretical phenomena as the "*fingerprint of chaos*". Notice that for this to occur in the output of the BSEV the sample size and number of iterations need to be very large. If we consider cylinder views of a set after several iterations of $\mathcal{F}$, such as in Figure 5.9, we can see the fingerprint start to emerge. Notice that such a visualization is not possible without the BSEV. The viewers mentioned in Chapter 3 are unable to allow the visualization of this behavior.

In the next section we present various plots produced by the BSEV and discuss how to interpret the plots to see where the Bunimovich stadium exhibits stronger and weaker sensitive dependencies.

[1] Here we mean the actual collision map, not the computer representation.

5.2 Visualizing Sensitive Dependence

In this section we discuss how to use plots produced by the BSEV to see where the Bunimovich stadium system has stronger and weaker sensitive dependences. We also make the argument that traditional viewers can not be used to visualize this behavior. In the process of our discussion we state how to minimize overlapping by modifying inputs to the BSEV.

As stated, the goal of this work is to produce software that allows users to see where sensitive dependence is strong and weak in the Bunimovich stadium. The software produced to meet this goal is the *Bunimovich Stadium Evolution Viewer* (BSEV). To see how we can use the BSEV to see where sensitive dependence is greater, consider the portion of trajectories for the collision points depicted in Figure 5.10. The plots in Figure 5.10 were generated with the inputs shown below.

1. Constant variable: phi
2. Value of conastant variable: $0.3926990817 \approx \frac{\pi}{8}$
3. Mean value of sample variable: $5.8904862255 \approx \frac{15\pi}{8}$
4. Sampled variable range: $pi/16$
5. Number of samples: 200
6. Sampling method: even
7. Number of iterations: 10
8. First iteration to display: 0

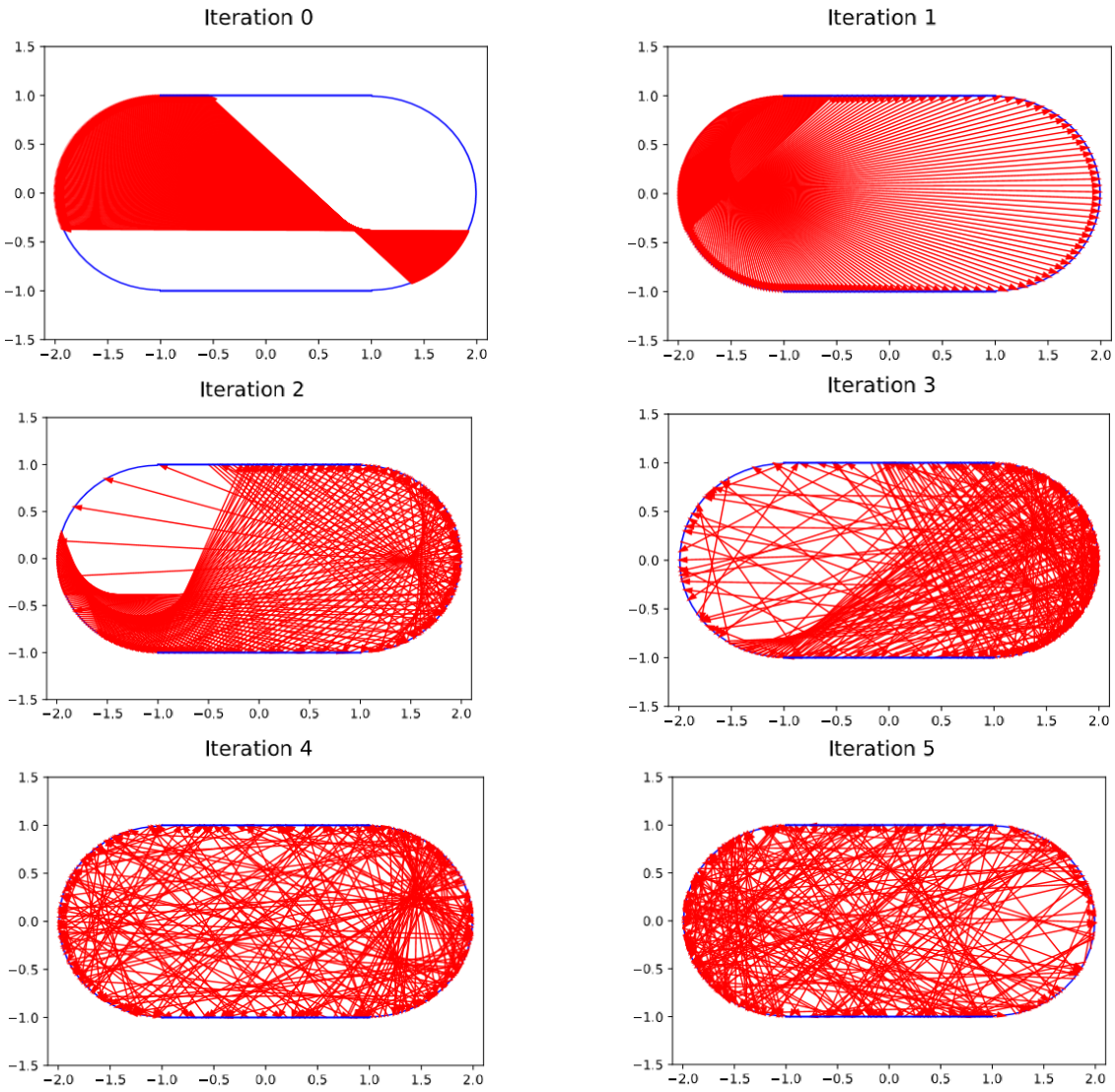

Figure 5.10: A portion of the trajectory of 200 collision points in the stadium (corresponds to the inputs listed on page 60)

In Figure 5.10 we can see that the collision points starting at "iteration 0" become distributed across the collision space $\mathcal{M}$ by iteration 5. However, it is hard to see where the sensitive dependence is greatest for this input. If we look at the corresponding cylindrical and spherical representations we can see where the system is more sensitive to initial conditions for the set of collision points in "iteration 0". The corresponding cylindrical plot is shown in Figures 5.11, though the cylindrical plots are easier to read, we also include the corresponding spherical plots of Figure 5.12. In the cylindrical plots, as well as the spherical ones, we can see that each image is composed of line segments of various colors. The end points of each segment correspond to two adjacent collision points that were sampled from the input range. As these two collision point's trajectories diverge over iterations, the line segment grows over the iterations. So, during any given iteration,

smaller line segments correspond to slower divergence, and larger ones correspond to faster divergence. In other words, the sensitive dependence is greater where the line segments are larger and smaller where the line segments are smaller. Notice that this behavior can not be observed in the viewers from Chapter 3. Only the novel visualization provided by the BSEV can allow us to *see divergence* in this manner. It is the cylinder view which enables us to visualize the growth of these line segments and thus infer information about divergence.

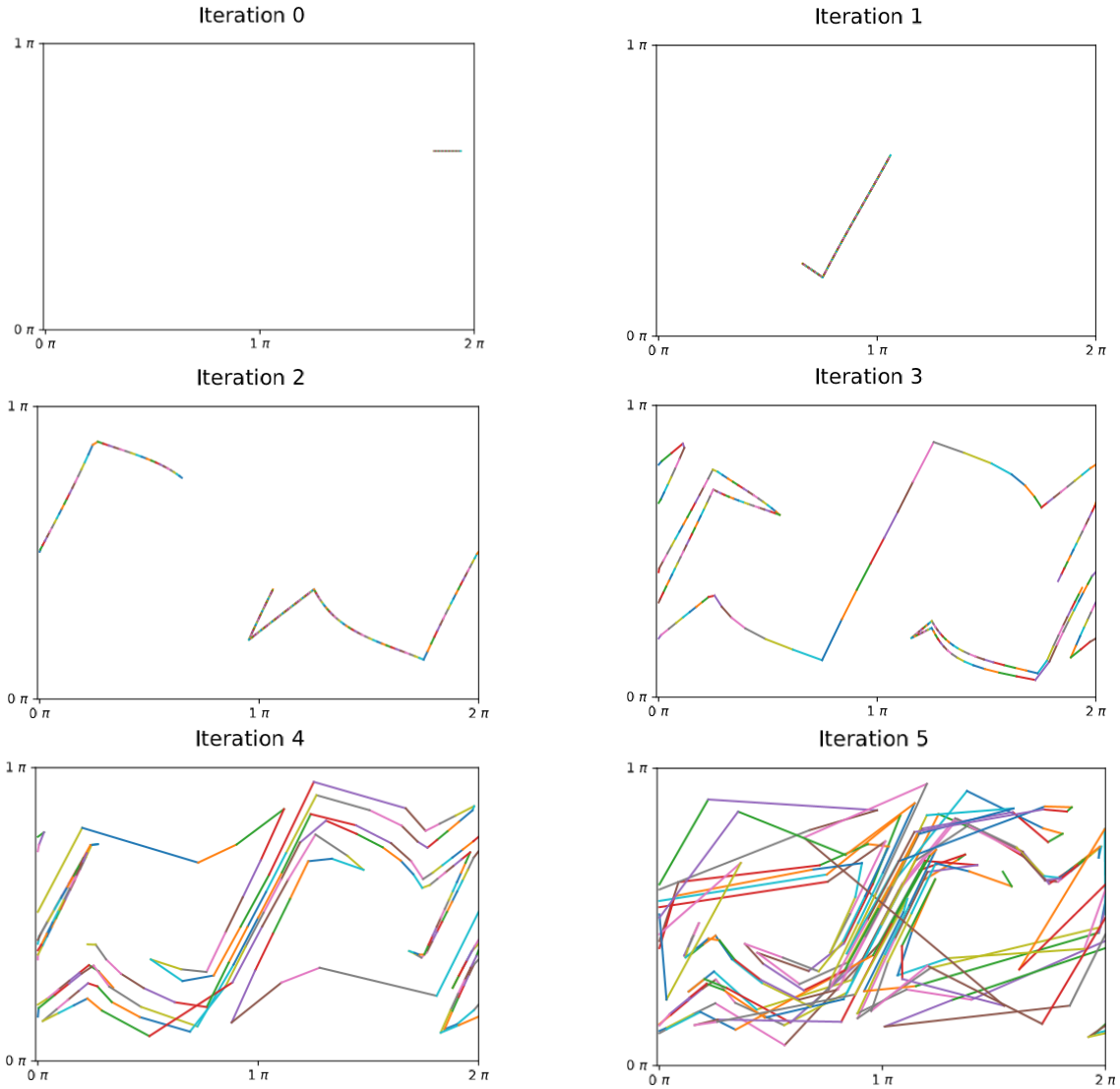

Figure 5.11: These cylindrical plots correspond to those of Figure 5.10

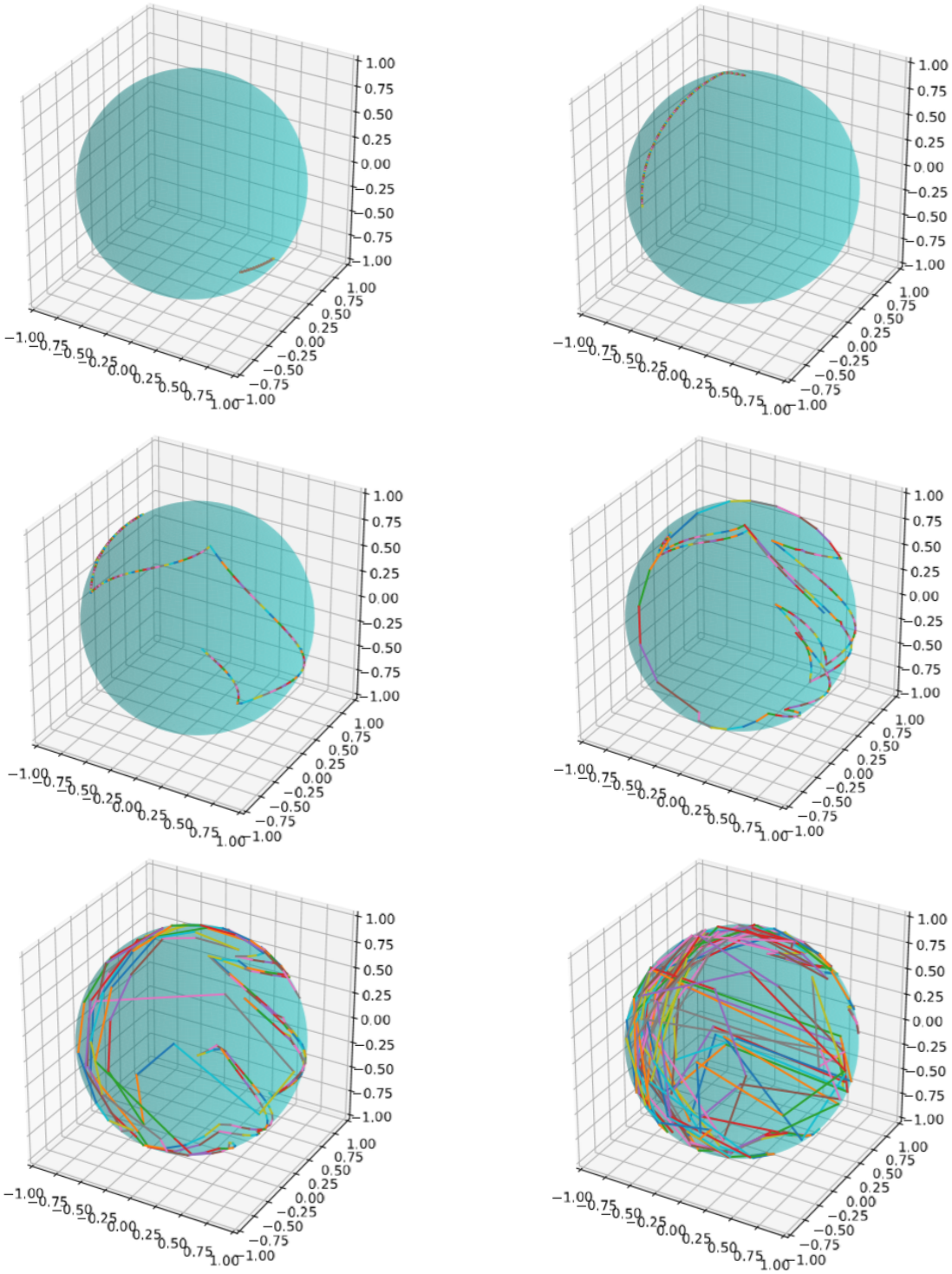

Figure 5.12: These spherical plots correspond to those of Figure 5.10

In Section 2.4 we saw that increasing the number of samples gave more representative plots by reducing intersections and effectively increasing the resolution. In the cylinder plots shown in Figure 5.11 and the corresponding spherical plots of Figure 5.12 there is significant intersecting/overlapping. In the cylinder

plots of Figures 5.13 and 5.14 the spherical plots of Figures 5.15 and 5.16 we have increased the number of samples to get a better resolution and included more iterations of the BSEV collision map. In addition, we modified the input "Sampled variable range" to $\frac{\pi}{64}$ to analyze a smaller range of collision points. During the first four iterations of the plots with $10,000$ samples it is difficult to see the the line segments because there are so many collision points sampled in a small area. At the fifth iteration, shown in Figure 5.13, we can start distinguishing the line segments and thus get information about sensitive dependence. Beyond iteration five we can see clearly where the system has more sensitivity; precisely where the line segments are longer. Because of the BSEV's ability to show the cylinder view we can visualize where the system has stronger sensitivity. Notice that in viewers such as those in Chapter 3 it would be very difficult or impossible to infer this kind of information about sensitivity.

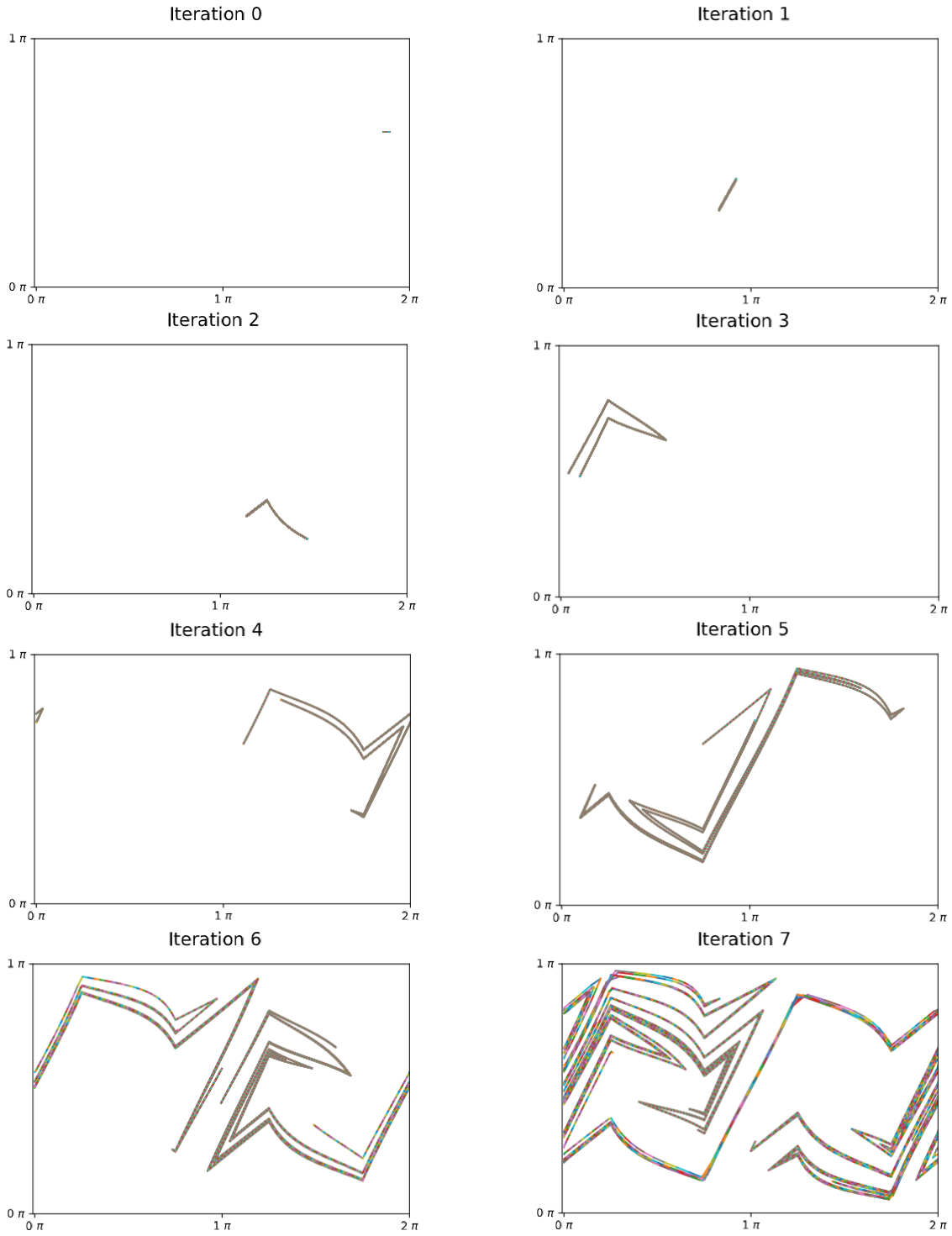

Figure 5.13: These cylindrical plots correspond to those of Figure 5.11 except that there are $10,000$ samples from a restricted sample range (up to iteration 7)

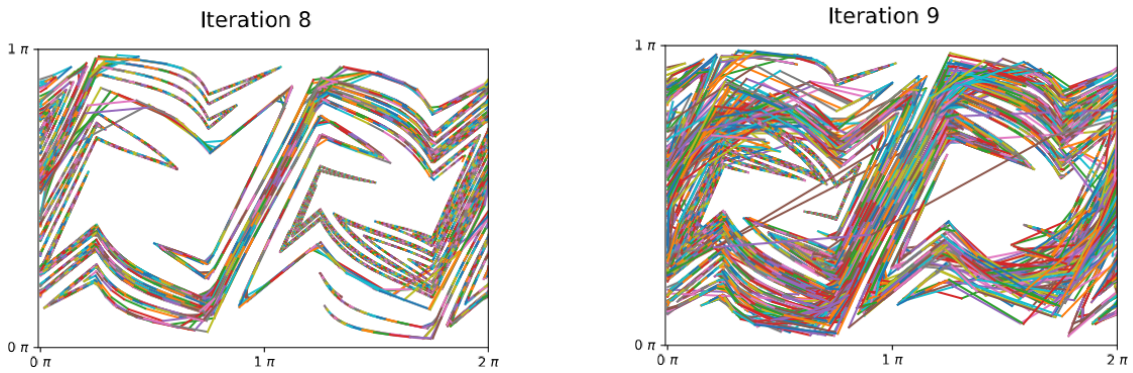

Figure 5.14: These cylindrical plots correspond to those of Figure 5.11 except that there are $10,000$ samples from a restricted sample range (iterations 8 and 9)

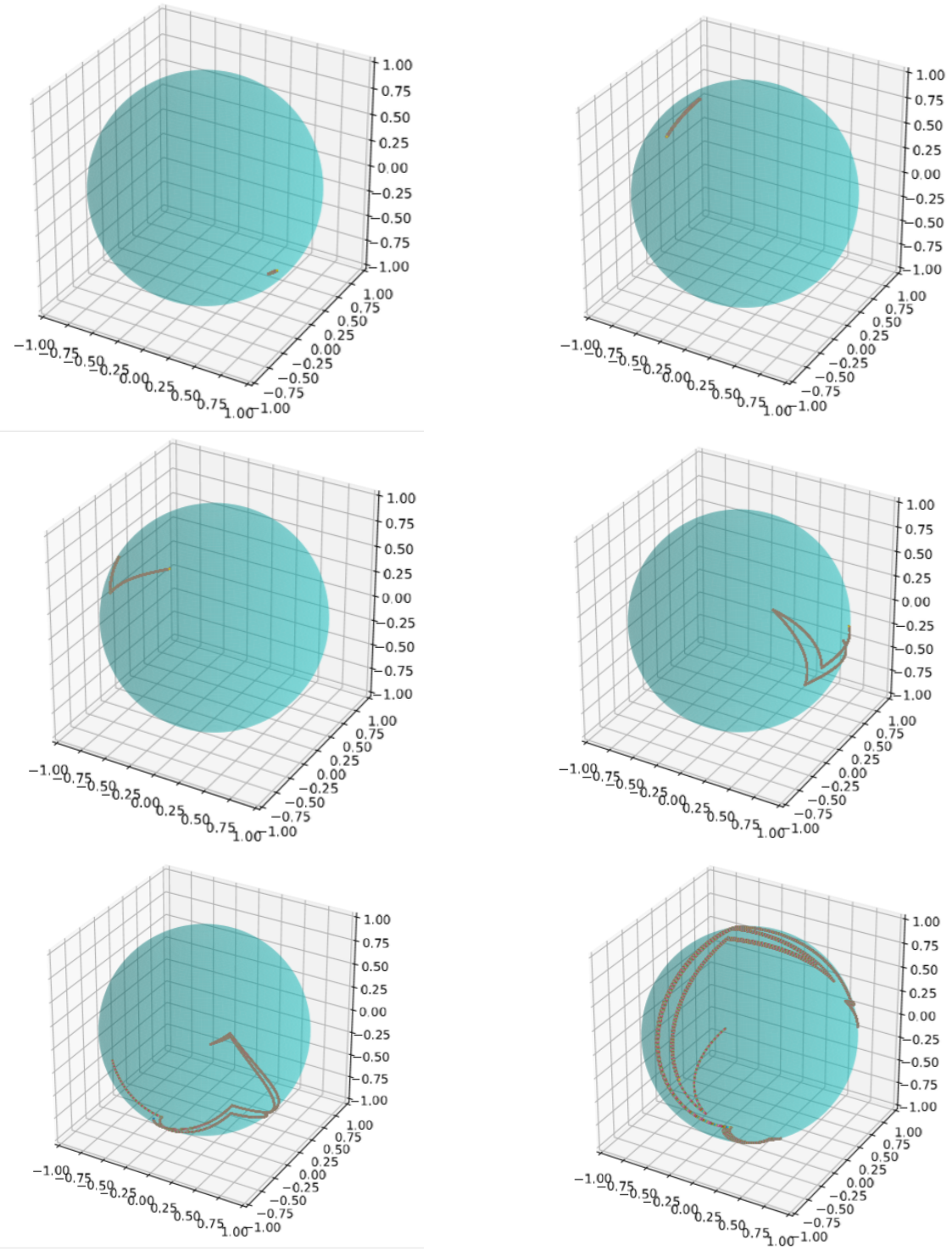

Figure 5.15: These spherical plots correspond to those of Figure 5.12 except that there are $10,000$ samples from a restricted sample range (up to iteration 5)

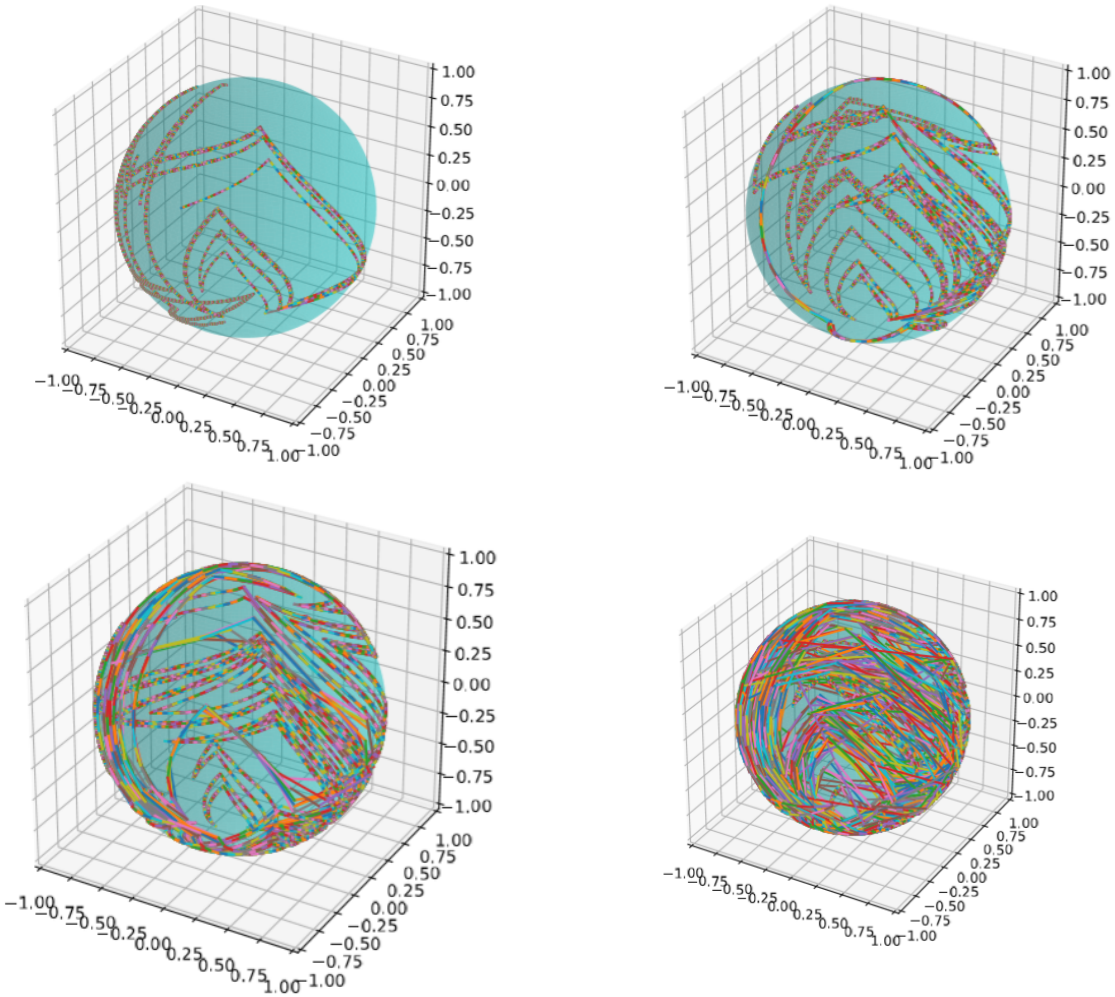

Figure 5.16: These spherical plots correspond to those of Figure 5.12 except that there are $10,000$ samples from a restricted sample range (iterations 6 through 9)

Earlier in this chapter we mentioned that inputs could be modified to make sure that we could prolong the number of iterations without intersections. We accomplish this by shrinking the range in which the non-constant variable varies. We have seen examples of restricting the range in Figures 5.13 through 5.16.In Figures 5.17 through 5.20, the same inputs were supplied to the BSEV listed on 60, with outputs shown in Figures 5.13 through 5.16, except that the range of θ values is smaller than the range in the previous Figures. So the input labeled "Sampled variable range" is $\frac{\pi}{256}$ instead of $\frac{\pi}{16}$. Restricting the range in this way also allows us to "zoom" in and see the evolution of the system with collision points from a smaller range. The resulting plots are easier to read because of less clutter.

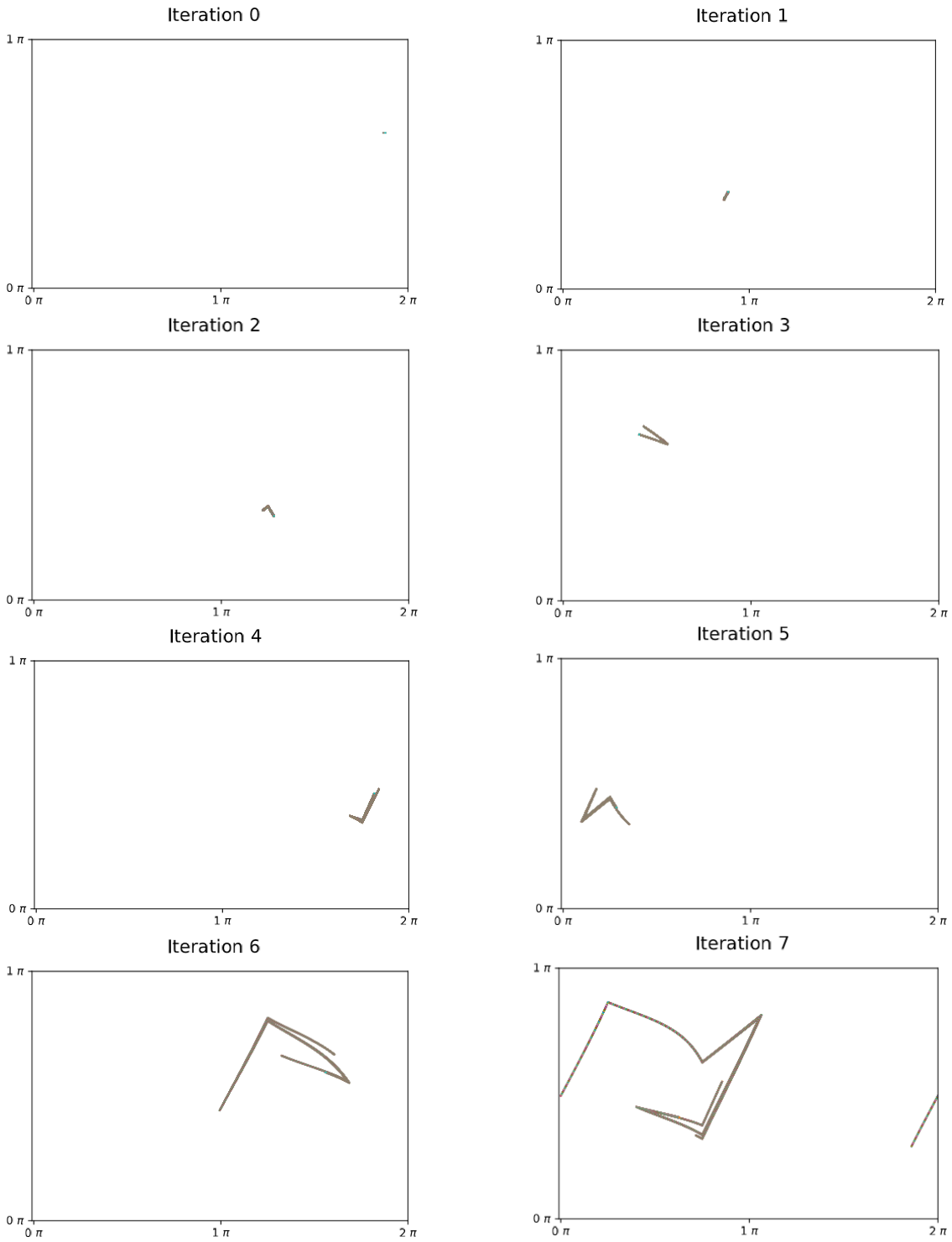

Figure 5.17: These cylindrical plots correspond to those of Figure 5.13 except that the samples are from a more restricted range (up to iteration 7)

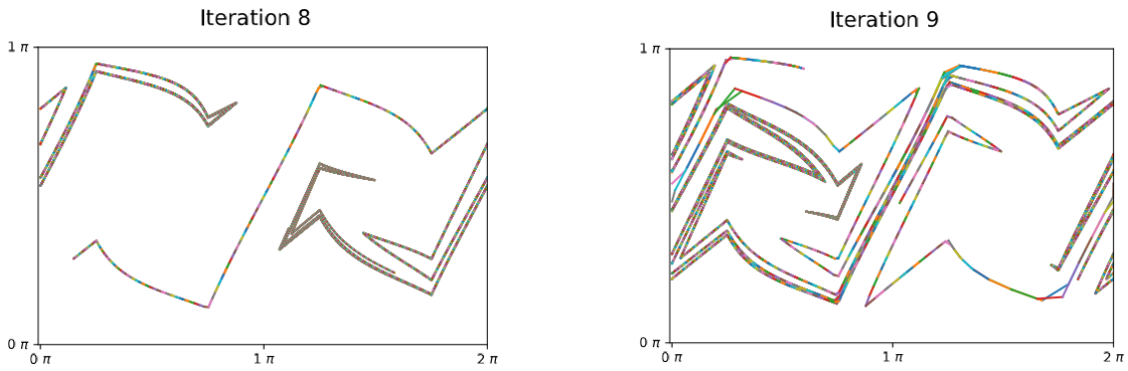

Figure 5.18: These cylindrical plots correspond to those of Figure 5.14 except that the samples are from a more restricted range (iterations 8 and 9)

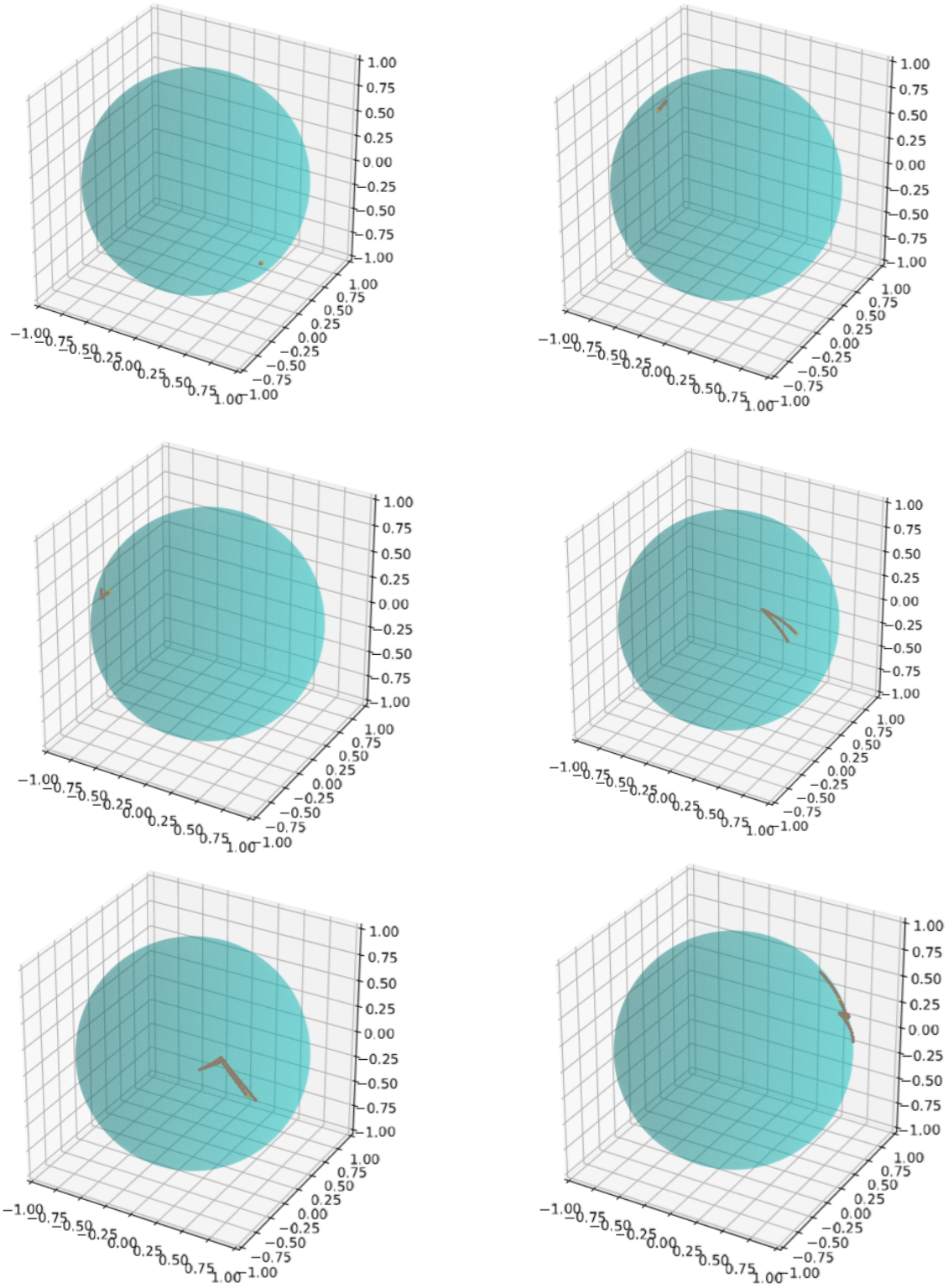

Figure 5.19: These spherical plots correspond to those of Figure 5.15 except that the samples are from a more restricted range (up to iteration 5)

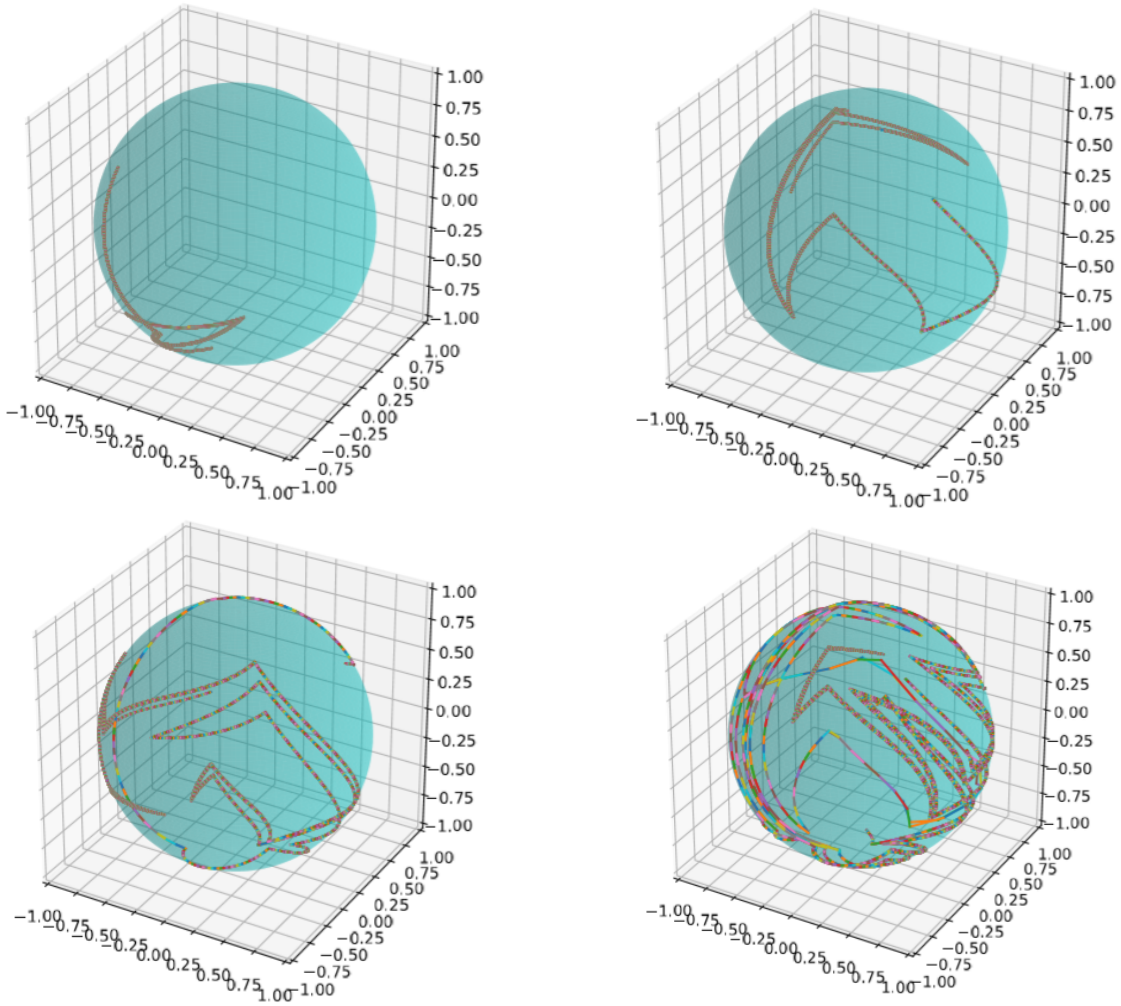

Figure 5.20: These spherical plots correspond to those of Figure 5.16 except that the samples are from a more restricted range (iterations 6 through 9)

We mentioned that we can use the plots produced by the BSEV to see for which inputs sensitive dependence is stronger and for which its weaker. As an example of an input with little sensitive dependence, consider the plots in Figures 5.21 and 5.22. In this example divergence occurs very slowly because the billiards moving along these trajectories bounce along flat sides for long periods. The corresponding cylinder plots are shown in Figures 5.25 and 5.27. The divergence is so slow that a sample size of 20 gave a high enough resolution to ensure no intersections. Because the divergence is slow the line segments grow slowly. When compared to Figure 5.11 the line segments grow far slower. Therefore, we can conclude that for this input the Bunimovich stadium less sensitive to initial conditions. Notice that the slow divergence can be seen in the stadium view. This means that viewers like those in Chapter 3 would be able to show where divergence is small. After many iterations, however, divergence will start to become stronger even for those trajectories with slow divergence. It is at this point that the traditional viewers are no longer helpful for

visualizing divergence and the BSEV is required.

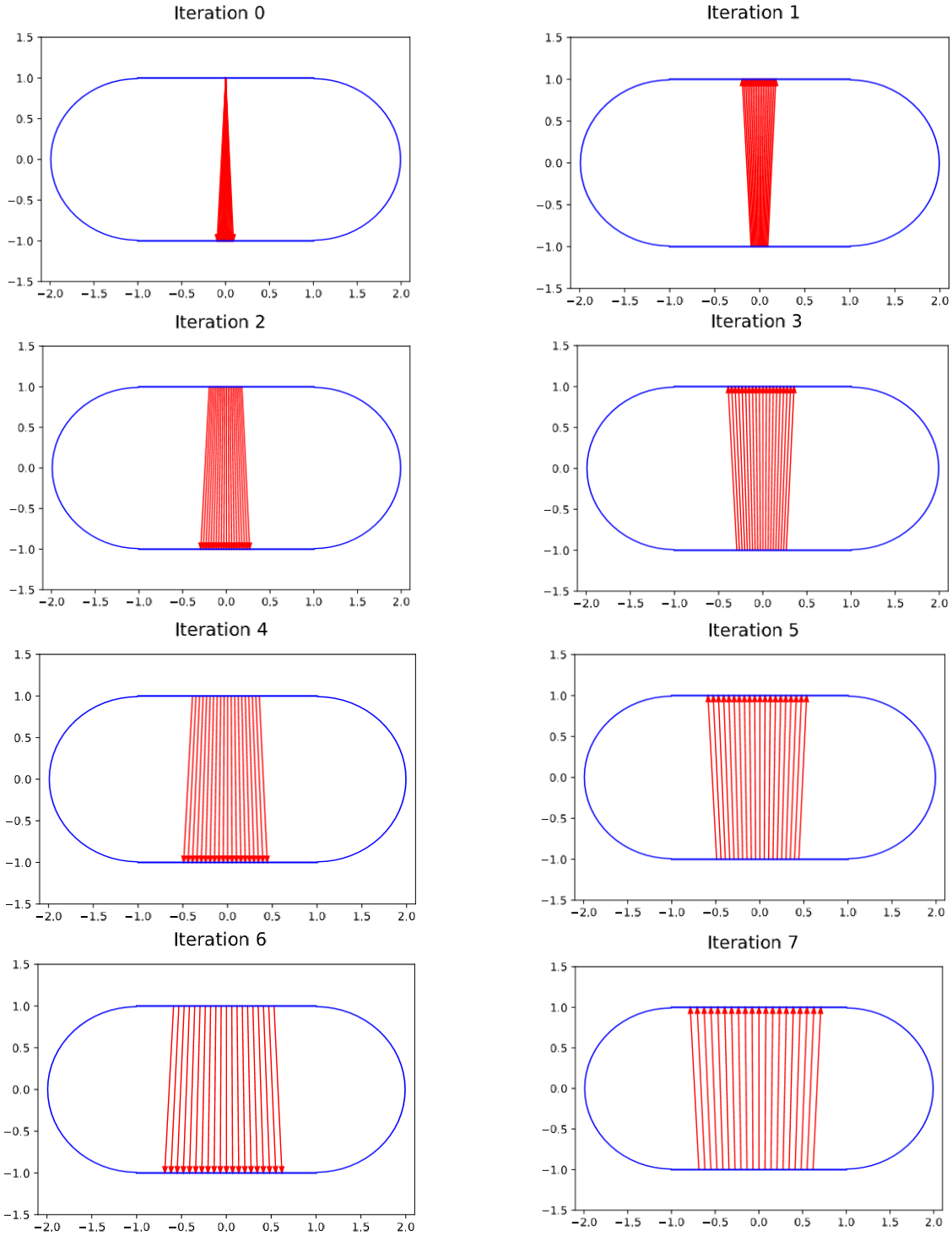

Figure 5.21: Portion of a trajectory with slow divergence (first 7 iterations)

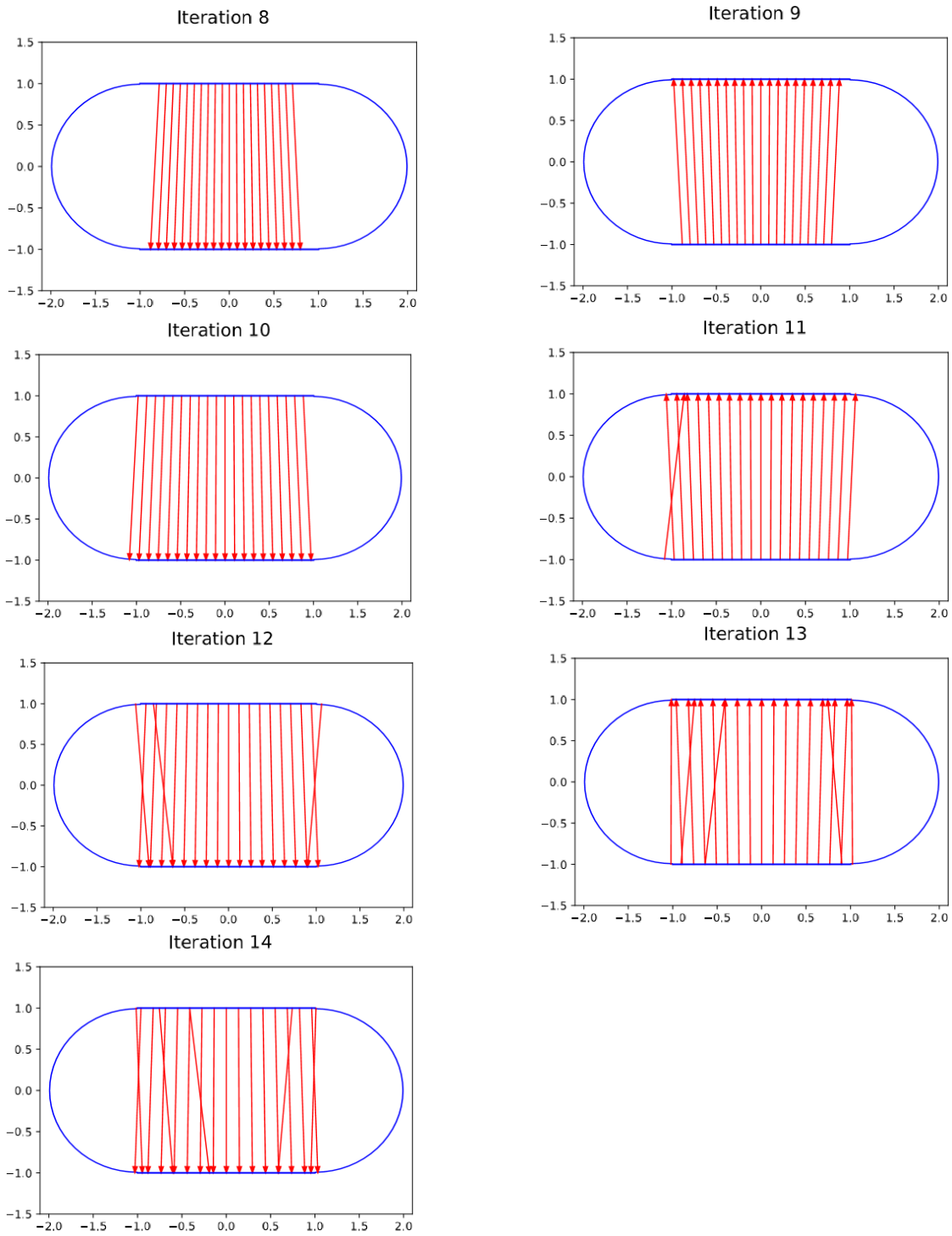

Figure 5.22: Portion of a trajectory with slow divergence (iterations 8 through 14)

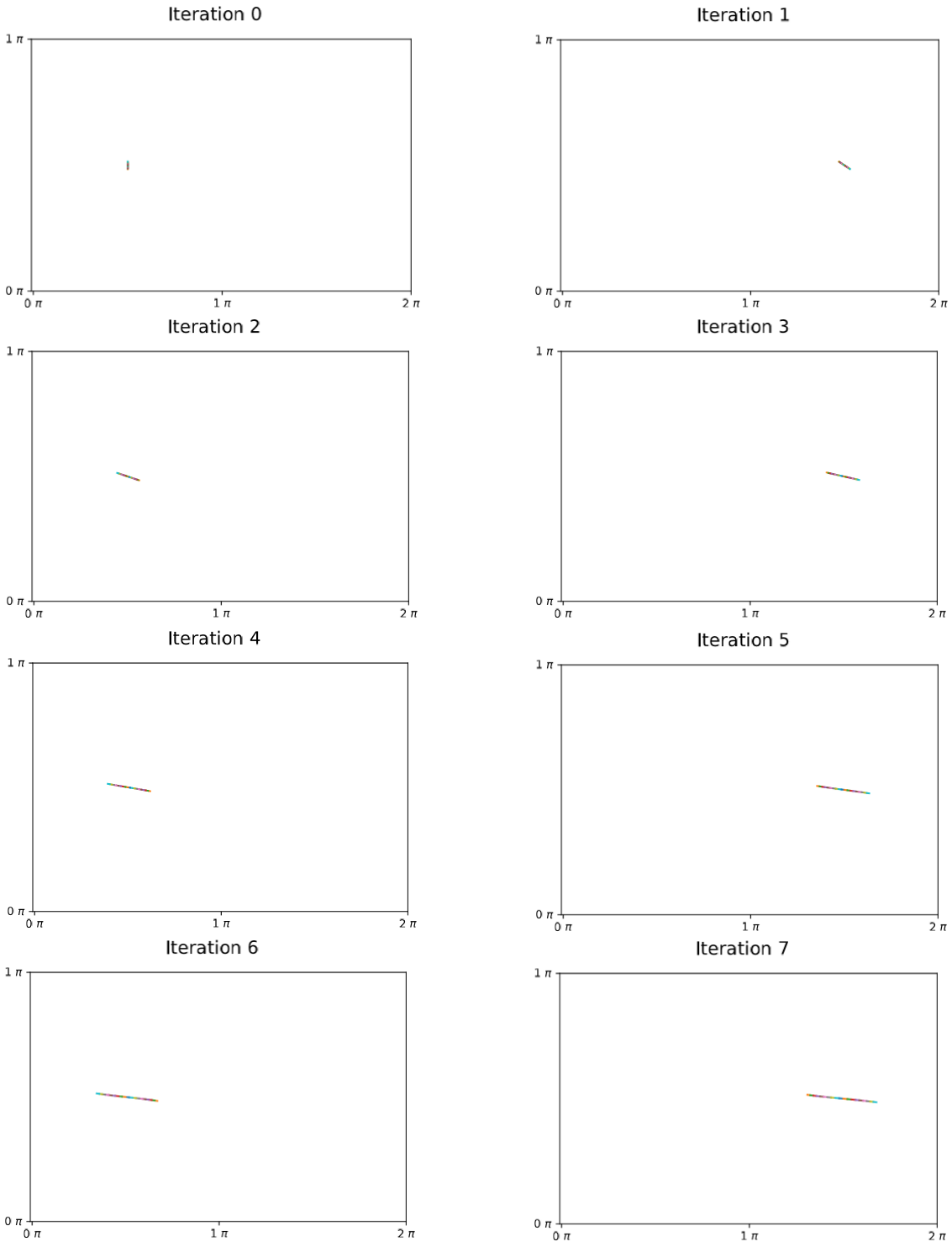

Figure 5.23: Cylinder plots of the portion of a trajectory with slow divergence from Figure 5.21 (first 7 iterations)

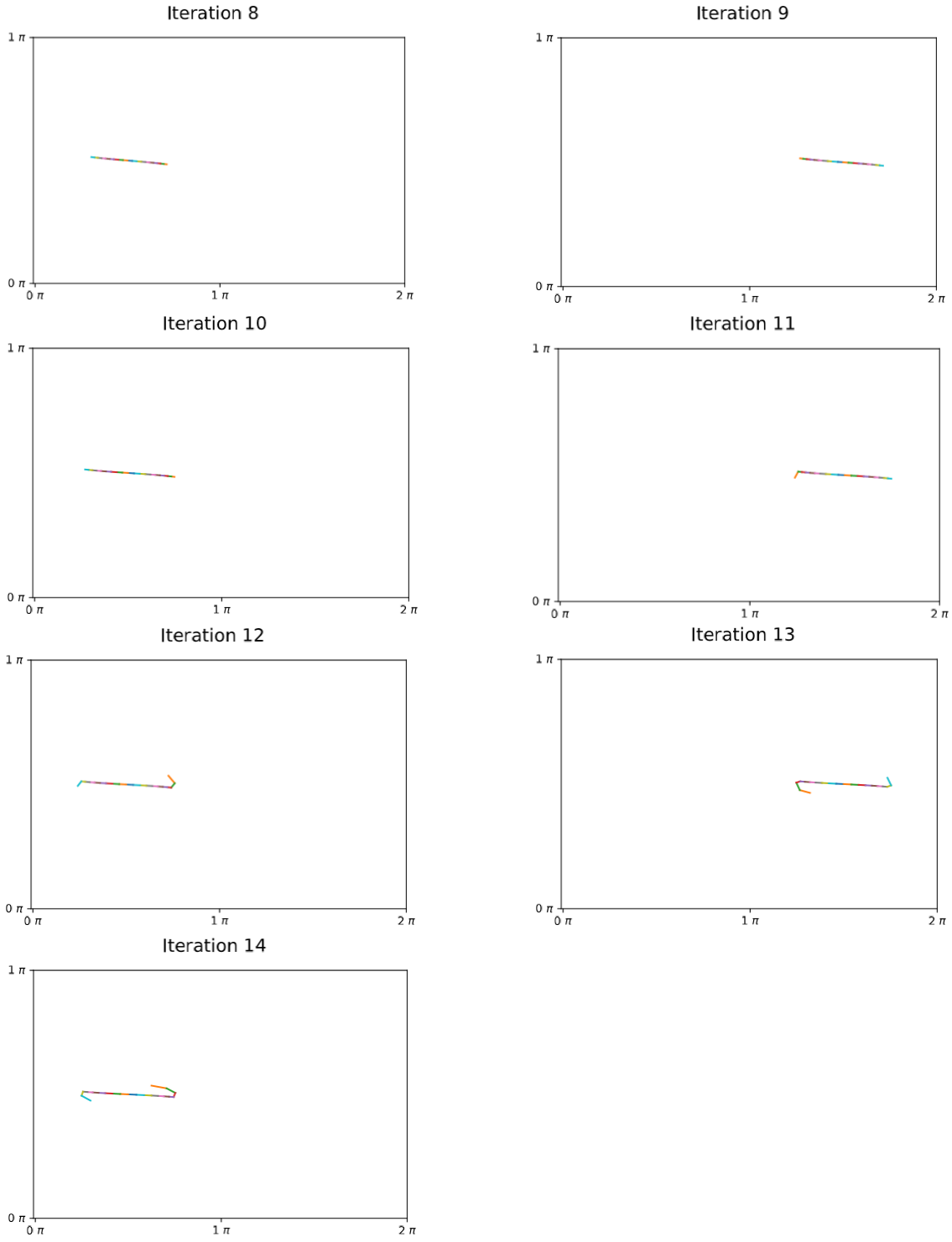

Figure 5.24: Cylinder plots of the portion of a trajectory with slow divergence from Figure 5.22 (iterations 8 through 14)

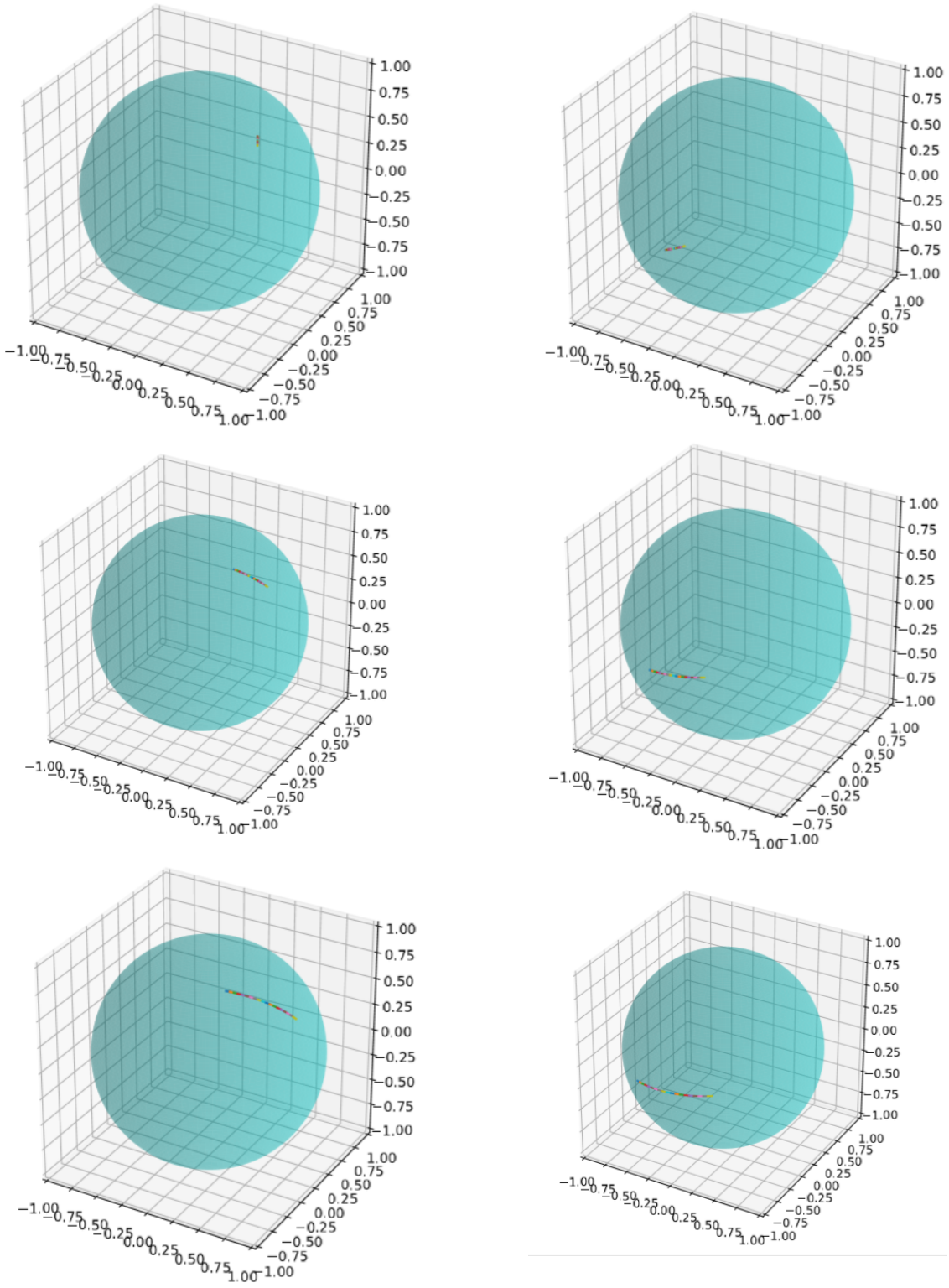

Figure 5.25: Spherical plots of the portion of a trajectory with slow divergence from Figure 5.21 (first 5 iterations)

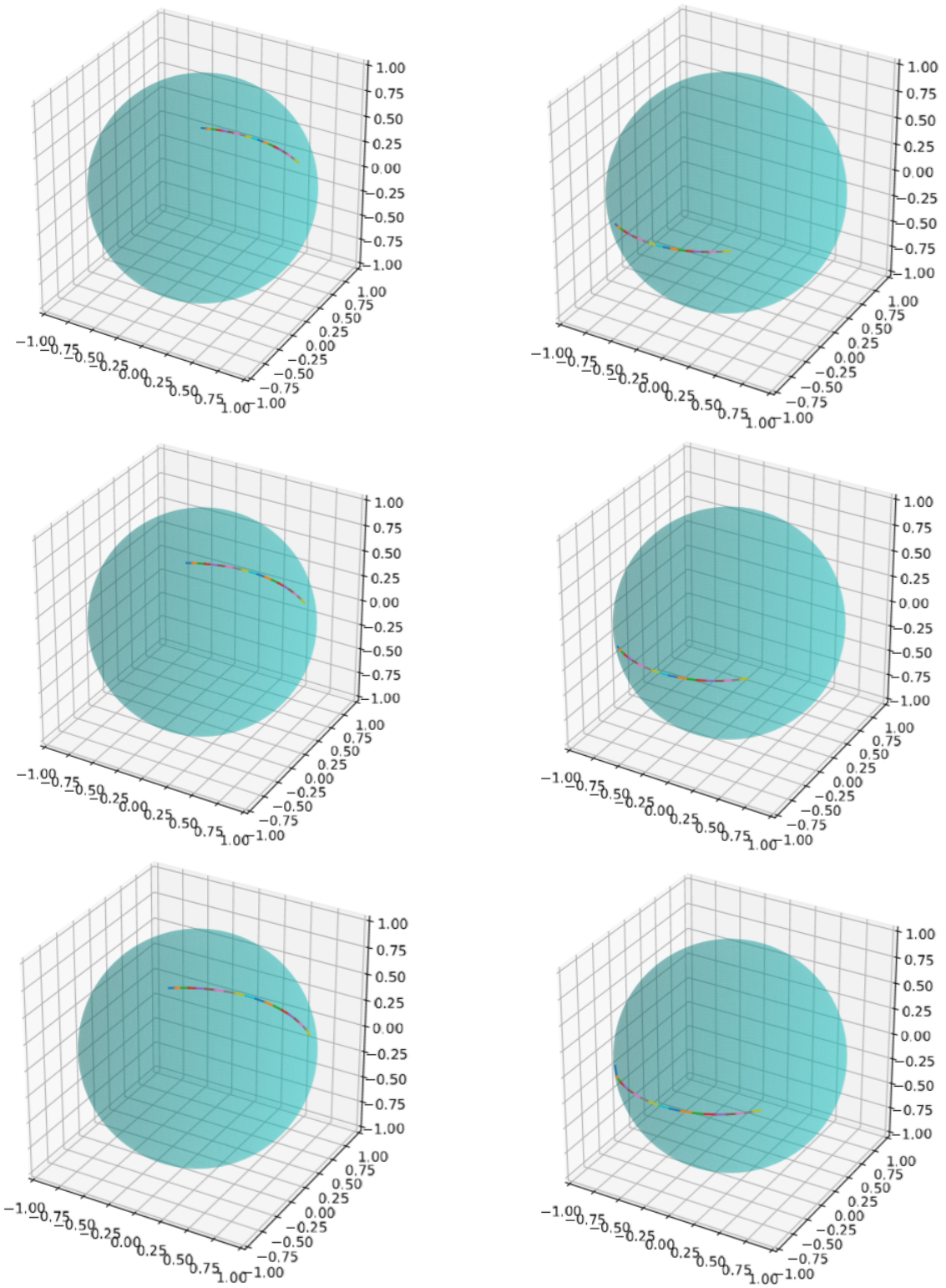

Figure 5.26: Spherical plots of the portion of a trajectory with slow divergence from Figure 5.22 (iterations 6 through 11)

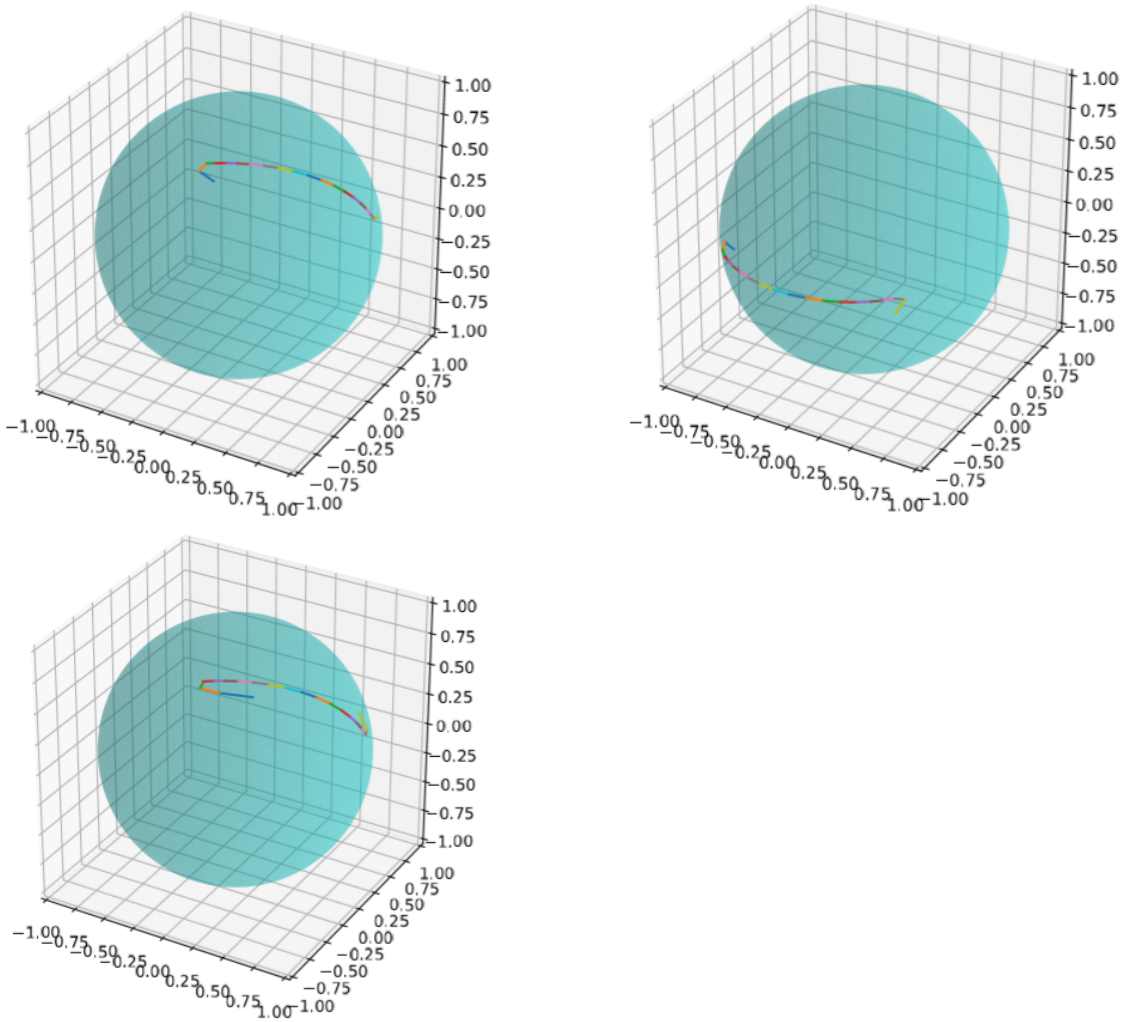

Figure 5.27: Spherical plots of the portion of a trajectory with slow divergence from Figure 5.22 (iterations 12 through 14)

We conclude this chapter by summarizing the uses of the *Bunimovich Stadia Evolution Viewer* (BSEV) software to visualize sensitive dependence. Mainly, the amount of divergence and thus sensitive dependence is proportional to the rate at which line segments grow in the cylinder and spherical views. When the segments grow slowly the divergence is slow and so the sensitive dependence is small. When the segments grow quickly, the divergence is fast and thus the sensitive dependence is great. Thus, with the BSEV we can gain insights into where the Bunimovich stadium billiard is more sensitive or less sensitive to initial conditions. We reiterate that the insights which are easily inferred using the BSEV are difficult or impossible to infer using other viewers. Since sensitive dependence is an important behavior of chaos the BSEV can help us gain insight into the chaotic behavior of the Bunimovich stadium. The chaotic nature of the Bunimovich stadium is the reason for how intricate the patterns produced by the software are. We encourage the reader to try many different inputs and see what intricate images may result.

Chapter 6

Conclusion

The Bunimovich stadium is a chaotic dynamical system first described by Bunimovich. This actively researched system has been of interest to mathematicians and physicists since it was discovered to be chaotic. Its chaotic behavior disappears when the lengths of the flat walls become 0 otherwise it remains. Because the system is chaotic it is sensitive to initial conditions. For certain initial conditions, divergence is slow and for others it is very fast. Previous visualization tools for the system have been created so we can see the system's evolution. These viewers only show billiards moving within the stadium itself which makes it difficult to visualize sensitive dependence. The BSEV does not have these shortcomings. By taking advantage of the fact that the collision space of the Bunimovich stadium can be viewed as a cylinder, we have created a computer model to view the system's evolution in a novel way. In the cylinder representation, sets of close collision points correspond to line segments. When applying the collision map to such sets using the BSEV we can see the system's evolution as was not possible in the viewers of Roux and Scheidegger. When the line segments connecting two collision points grows from iteration to iteration, this corresponds to divergence of the two point's trajectories. When the line segments grow quickly, the divergence is fast, and for these inputs sensitive dependence is greater. When the segments grow slowly, divergence is slow, and for such inputs the sensitive dependence is lesser. It is the novel visualization of the cylinder view produced by the BSEV which allows one to visualize divergence and thus sensitivity. The aforementioned viewers lack this capability. Sensitivity to initial conditions is a key behavior of chaos and thus the BSEV may be able to provide new insights into the stadium's chaotic behavior. The BSEV software can be freely obtained at https://github.com/shoemarw/BunimovichStadium and instructions for its use are in Appendix B.

Chapter 7

Future Work

The *Bunimovich Stadia Evolution Viewer* (BSEV) is freely available via:

https://github.com/shoemarw/BunimovichStadium

The BSEV software produced all of the plots in Chapters 4 and 5. It is useful for visualizing the sensitive dependence of the Bunimovich stadium Billiard. Here, we suggest some potential improvements of the software and future avenues of research in the vein of this work.

The graphical user interface for the BSEV is restrictive with respect to the inputs that it accepts from users. Potential improvements should allow the user to supply more custom inputs such as the user should be able to supply their own numeric value for the "sampled variable range" field. The user should also have more control over the number of points sampled from the range of input values. The user should also be able to specify more iterations than are currently available.

A key improvement in the performance of the software would involve "smart" sampling. The software currently samples points that are evenly spaced in the input set. A better way to sample points would be to sample points where the divergence is greatest; this corresponds to where the derivative of the collision map $\mathcal{F}$ has the largest values. A "smart" sampling algorithm would take into account where the derivative of the map is greatest and sample more points in those regions. This would allow the software to give better resolution with fewer sampled points and improve the speed of the software.

The software can also be improved by modifying the stadium view outputs. When viewing a plot a set of sampled points in the stadium it is difficult to trace the trajectory of particular billiards. In order to make it easier to trace paths future software should allow the user to specify a particular collision point, or set of collision points, and include labels in the plots so the particular path(s) can be distinguished easily.

The reason this is not currently implemented in the BSEV is because it uses the library matplotlib to graph plots. Adding the aforementioned functionality to the BSEV would have required modification of the libraries' source code or scrapping the use of matplotlib all together.

Another avenue of future research would involve improving the accuracy of the BSEV and using the properties of the Bunimovich stadium billiard to justify the programs correctness. This involves something called the shadowing lemma, we refer the reader to [5] for more information. Briefly, if the shadowing lemma holds for a certain dynamical system then the accuracy of correctly implemented computer models of the system is almost guaranteed. The shadowing lemma holds for dynamical systems with uniformly hyperbolic maps. The Bunimovich stadium's collision map $\mathcal{F}$ is not uniformly hyperbolic, however, a restricted version of it $\mathcal{F}_\diamond$ is. We refer the interested reader to [5] for a definition of $\mathcal{F}_\diamond$.

Appendix A

Code for the Bunimovich Stadia Evolution Viewer Software

A.1 BunimovichStadiaEvolutionGUI.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan  3 13:56:14 2019
Stadia evoulution Viewer (GUI). Provides a GUI for the user to input
the parameters of the system. Enables the Stadia View, Cylinder View,
and Spherical View. Uses Plotter.py
@author: Randy
"""

import Tkinter as tk
from Tkinter import StringVar, Label, Entry, OptionMenu, Menu, Message, Button
from MessageText import TEXT, ITERMESSAGE, PHIMESSAGE1, PHIMESSAGE2
from Plotter import plotter
from CoordinateConversion import mod2pi
import math


pi = math.pi
```

```
# Create a GUI window
master = tk.Tk()

## Create global variables (those in drop down menus) ##
constvar   = StringVar(master) # Which variable (theta/phi) is constant?
eRange     = StringVar(master) # Specifies a window for the other var to vary
samples    = StringVar(master) # Specifies the number of samples
sampleType = StringVar(master) # Specifies the sampling technique to be used
iterations = StringVar(master) # Specifies the number of iterations
start      = StringVar(master) # Specifies the iteration to start displaying

## Global entry variables
const = StringVar(master)
var   = StringVar(master)

## Global check box variables
c_check = StringVar()
s_check = StringVar()
b_check = StringVar()

# Set default values for the global variables (displayed on drop-downs)
constvar.set("constant variable")
eRange.set("sampled variable range")
samples.set("samples")
sampleType.set("sample method")
iterations.set("iterations")
start.set("first iteration")
# Make sure check boxes start unchecked
c_check.set(0)
s_check.set(0)
b_check.set(0)
```

```
## A function for displaying the "about" information under the help menu ##
## This uses the variable TEXT imported from helpAboutText.py ##
def about():
    help_window = tk.Toplevel(master)
    help_window.geometry("1000x1000")
    help_window.title("About the Bunimovich Stadia Evolution Viewer")

    text = Message(help_window, text = TEXT, padx = 100)
    text.pack()

## A function for resetting all values to their default ##
def reset_all():
    # Clear the text entry boxes
    constVar_field.delete(0, tk.END)
    samplVar_field.delete(0, tk.END)
    constvar.set("constant variable")
    eRange.set("sampled variable range")
    samples.set("samples")
    sampleType.set("sample method")
    iterations.set("iterations")
    start.set("first iteration")
    c_check.set(0)
    s_check.set(0)
    b_check.set(0)


## A function for ensuring the generate button can only be pressed when ##
## all fields have been filled in. ------------------------------------ ##
def protectGenerate(*args):
    a = not (constvar.get()    == "constant variable")
    b = not (eRange.get()      == "sampled variable range")
```

```
    c = not (samples.get()     == "samples")
    d = not (sampleType.get() == "sample_method")
    e = not (iterations.get() == "iterations")
    f = not (start.get()       == "first_iteration")
    g = const.get()
    h = var.get()
    i = c_check.get() == '1' or s_check.get() == '1' or b_check.get() == '1'
    if a and b and c and d and e and f and g and h and i:
        generate_button.config(state = 'normal')
    else:
        generate_button.config(state = 'disabled')

## A function for ensuring that the text boxes supplying the values of ##
## theta and phi are float ------------------------------------------- ##
def validate(action, index, value_if_allowed, prior_value, text, \
             validation_type, trigger_type, widget_name):
    # action=1 -> insert
    if(action=='1'):
        if text in '0123456789.-+':
            try:
                float(value_if_allowed)
                return True
            except ValueError:
                return False
        else:
            return False
    else:
        return True

## A function for generating the visual output
def generate():
    ## Get the values of all variables ##
```

```
cvar = constvar.get()          # Which variable is the constant one?
sams = int(samples.get())      # Number of samples
avg  = float(var.get())        # Average value of sampled variable
eps  = eRange.get()            # Range of sampled variable
para = float(const.get())      # Value of constant variable
its  = int(iterations.get()) # Number of iterations
star = int(start.get())        # First iteration to display to user
styp = sampleType.get()        # Type of sampling method

## Set the view variables according to the check boxes make their default
## values an empty string.
c,s,b = "","",""
if c_check.get() == '1':
    c = "c"            # Indicates if cylinder view is checked.
if s_check.get() == '1':
    s = "s"            # Indicates if spherical view is checked.
if b_check.get() == '1':
    b = "b"            # Indicates if stadia view is checked.

# get the eps value ready to pass to the plotter
if eps == "pi/4":
    epsi = pi/4
elif eps == "pi/16":
    epsi = pi/16
elif eps == "pi/64":
    epsi = pi/64
elif eps == "pi/256":
    epsi = pi/256
elif eps == "pi/1024":
    epsi = pi/1024
else:
    print "Invalid input from drop down menu"
```

```
    ## Make sure that the number of iterations and the first iteration to ##
    ## display are consistent ------------------------------------------- ##
    if its <= star:
        iteration_error_window = tk.Toplevel(master)
        iteration_error_window.geometry("300x150")
        iteration_error_window.title("ILLEGAL INPUT DETECTED")

        text = Message(iteration_error_window, text = ITERMESSAGE)
        text.pack()
        return False

    ## Make sure that theta is in the proper range ##
    if cvar == "theta" and (para < 0 or para >= 2*pi):
        # para is a theta value and its outside of the [0,2pi) range.
        # Mod it into the [0,2pi) range.
        para = mod2pi(para)



    ## Make sure that phi is in the proper range ##
    if cvar == "phi" and (para <= -pi/2 or para >= pi/2):
        phi_error_window = tk.Toplevel(master)
        phi_error_window.geometry("400x300")
        phi_error_window.title("ILLEGAL INPUT DETECTED")

        text = Message(phi_error_window, text = PHIMESSAGE1)
        text.pack()
        return False
    ## Make sure that phi is in the proper range ##
    if cvar == "theta" and (avg - epsi <= -pi/2 or avg + epsi >= pi/2):
        phi_error_window = tk.Toplevel(master)
        phi_error_window.geometry("400x200")
```

```
        phi_error_window.title("ILLEGAL INPUT DETECTED")

        text = Message(phi_error_window, text = PHIMESSAGE2)
        text.pack()
        return False

    if plotter(cvar, sams, avg - epsi, avg + epsi, para, its+1, star, 2, \
                    styp, c + s + b):
        confirm_window = tk.Toplevel(master)
        confirm_window.geometry("200x100")
        confirm_window.title("Success!")

        msg = """
        Pdf(s) of the desired images were created
        in the folder containing this application.
        """
        text = Message(confirm_window, text = msg, padx = 10)
        text.pack()

if __name__ == "__main__":

    # Set the size of the GUI window
    master.geometry("460x285")
    master.title("Bunimovich Stadia Evolution Viewer")

    ## Create a Menu ##
    menu = Menu(master)
    master.config(menu = menu)

    # Add help menu with an about option
    helpmenu = Menu(menu)
    menu.add_cascade(label="Help", menu=helpmenu)
```

```
helpmenu.add_command(label="About...", command=about)

## WIDGETS ##
# Labels for drop downs
label_const    = Label(master, text = "Constant Variable: ")
label_eRang    = Label(master, text = "Sampled Variable Range: ")
label_samples = Label(master, text = "Number of Samples: ")
label_samType = Label(master, text = "Sampling Method: ")
label_iters    = Label(master, text = "Number of Iterations: ")
label_startIt = Label(master, text = "First Iteration to Display: ")

# Lables for text input boxes
label_value_const    = Label(master, text = "Value of Constant Variable: ")
lable_value_sampVar = Label(master, text = \
                            "Mean Value of Sampled Variable: ")

## Use the grid method to place the widgets ##
label_const.grid(row = 0, column = 0)
label_value_const.grid(row = 1, column = 0)
lable_value_sampVar.grid(row = 2, column = 0)
label_eRang.grid(row = 3, column = 0)
label_samples.grid(row = 4, column = 0)
label_samType.grid(row = 5, column = 0)
label_iters.grid(row = 6, column = 0)
label_startIt.grid(row = 7, column = 0)

## Create text entry boxes for getting theta and phi ##
vcmd = (master.register(validate), '%d', '%i', '%P', '%s', '%S', '%v', \
                                    '%V', '%W')
constVar_field = Entry(master, textvariable = const, validate = 'key', \
                       validatecommand = vcmd)
samplVar_field = Entry(master, textvariable = var, validate = 'key', \
```

```
                validatecommand = vcmd)

# Place the text entry boxes on the window
constVar_field.grid(row = 1, column = 2, ipadx = "20")
samplVar_field.grid(row = 2, column = 2, ipadx = "20")

## Create the list of option for each drop down ##
constVar_list = ["theta", "phi"]
eRang_list = ["pi/4", "pi/16", "pi/64", "pi/256", "pi/1024"]
samples_list = ["2", "5", "10", "20", "50", "100", "200", "500", "1000",\
                "2000", "5000", "10000"]
samType_list = ["even", "random"]
iterations_list = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10", \
                   "11", "12", "13", "14", "15"]
start_list = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", \
              "11", "12", "13", "14", "15"]
## Create drop down menus ##
const_option    = OptionMenu(master, constvar, *constVar_list)
eRang_option    = OptionMenu(master, eRange, *eRang_list)
samples_option = OptionMenu(master, samples, *samples_list)
samType_option = OptionMenu(master, sampleType, *samType_list)
iterate_option = OptionMenu(master, iterations, *iterations_list)
start_option    = OptionMenu(master, start, *start_list)

# Place the drop down menus
const_option.grid(row = 0, column = 2)
eRang_option.grid(row = 3, column = 2)
samples_option.grid(row = 4, column = 2)
samType_option.grid(row = 5, column = 2)
iterate_option.grid(row = 6, column = 2)
start_option.grid(row = 7, column = 2)
```

```
## Create buttons ##
clear_button    = Button(master, text = "Reset All", command = reset_all)
generate_button = Button(master, text = "Generate", command = generate)
quit_button     = Button(master, text = "Quit", command = master.destroy)

# Place the buttons
quit_button.grid(row = 8, column = 0)
clear_button.grid(row = 8, column = 1)
generate_button.grid(row = 8, column = 2)

# Configure the generate_button to be disabled initially. It shall
# remain disabled until all fields are entered.
generate_button.config(state = 'disabled')

# Attach a trace to all variables. Use it to ensure that generate_button
# can only be pressed when all fields are entered.
constvar.trace("w", protectGenerate)
eRange.trace("w", protectGenerate)
samples.trace("w", protectGenerate)
sampleType.trace("w", protectGenerate)
iterations.trace("w", protectGenerate)
start.trace("w", protectGenerate)
const.trace("w", protectGenerate)
var.trace("w", protectGenerate)

## Create check buttons to provide options to the user
c_checkbutton = tk.Checkbutton(master, variable=c_check, \
                        text ='cylinder view', command=protectGenerate)
s_checkbutton = tk.Checkbutton(master, variable=s_check, \
                        text ='spherical view', command=protectGenerate)
b_checkbutton = tk.Checkbutton(master, variable=b_check, \
                        text ='stadia view', command=protectGenerate)
```

```
# Place check-buttons
c_checkbutton.grid(row = 9, column = 0)
s_checkbutton.grid(row = 9, column = 1)
b_checkbutton.grid(row = 9, column = 2)

master.mainloop()
```

A.2 Plotter.py

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan  5 12:42:50 2019
This code is used by the Bunimovich Stadia Evolution Viewer GUI to do plotting.
It produces plots in the cylindrical, spherical, and stadia views. Uses
ComputeIteration.py
@author: Randy
"""

import gc
from matplotlib.backends.backend_pdf import PdfPages
import matplotlib.pyplot as plt
import matplotlib.ticker as tck
import matplotlib.patches as mpatches
from mpl_toolkits import mplot3d
from ComputeIteration import image_const_phi
from ComputeIteration import image_const_theta
import math
import numpy as np

def plotter(const, samples, sampleParamLow, sampleParamHi, param, \
                    iterations, start, lam, sampleType, plotType):
    """ const          := is 'phi' (constant phi) xor 'theta' (constant theta).
        samples        := the number of samples of the parameter to be varied.
        sampleParamLow := lower bound for sampled values of the varied param.
        sampleParamHi  := upper bound for sampled values of the varied param.
        param          := the value of the constant parameter.
        iterations     := the number of iterations of the collision map.
        lam            := the ration of the stadias side to radius.
        sampleType     := the type of technique to use for sampling.
                          'even' for evenly spaced
```

```
                    'random' for uniformly random
    plotType        := which type of plot(s) is/are to be produced.
                    possible values:
                        'c'   for cylinder view only
                        's'   for spherical view only
                        'b'   for Bunimovich stadia view only
                        'cs'  for cylinder and spherical views
                        'cb'  for cylinder and stadia views
                        'sb'  for spherical and stadia views
                        'csb' for all three views

"""
pi = math.pi # We can always use some pi!

## Generate the data to be used in all plot views ##
if const == 'phi':
    # the constant parameter is phi, the varied parameter is theta.
    var = 'theta'
    images, cartesianImage = image_const_phi(sampleParamLow, \
            sampleParamHi, samples, param, iterations,  lam, sampleType)

elif const == 'theta':
    # the constant parameter is theta, the varied parameter is phi.
    var = 'phi'
    images, cartesianImage = image_const_theta(sampleParamLow, \
            sampleParamHi, samples, param, iterations,  lam, sampleType)
else:
    print "Invalid constant parameter."

print "Computation finished, preparing image..."

## Create a string to uniquely name the output pdf ##
```

```
    name = "const_" + const + "_" + str(param) + "_samples_" + str(samples) + \
           "_" + var + "_" + str(sampleParamLow) + "to" + str(sampleParamHi)
    name = "_" + name + "iters_" + str(iterations) + "_sType_" + sampleType

## ———————————————— CYLINDER VIEW ———————————————— ##
    # Check if the cylinder view is to be presented.
    if "c" in plotType:
        # Create a pdf for output
        ppC = PdfPages("cylinder" + name + ".pdf")
        # Proceed to present the cylinder view.
        for iteration in range(start ,iterations):
            fig = plt.figure(figsize=(6,4))
            ax = plt.subplot(111)
            plt.suptitle("Iteration " + str(iteration), fontsize=16)

            ax.yaxis.set_major_formatter(tck.FormatStrFormatter('%g $\pi$'))
            ax.yaxis.set_major_locator(tck.MultipleLocator(base=1.0))
            ax.xaxis.set_major_formatter(tck.FormatStrFormatter('%g $\pi$'))
            ax.xaxis.set_major_locator(tck.MultipleLocator(base=1.0))

            ax.set_xlim([-0.01, 2])
            ax.set_ylim([0, 1])
            ax.plot(0, 0, ms=0)

            points = images[iteration]
            # Go through the points plotting them as we go
            for i in range(1,samples):
                dist = abs(points[i-1][0] - points[i][0])
                # See if the regular distance is shorter than the compliment of
                # the distances.
                if dist < (2*pi - dist):
                    # "no wrapping"
```

```
            # The points must be plotted normally
            x = [points[i-1][0]/pi, points[i][0]/pi]
            y = [(points[i-1][1] + pi/2)/pi, (points[i][1] + pi/2)/pi]
            plt.plot(x, y)#3333333333333333333333333333333333333333333333
        else:
            # "wrapping"
            # The points must be plotted so that the line between them
            # goes the "other way" around the cylinder.
            # Get the average of the two psi values of the points.
            # Then shift up by pi/2 to turn the phi value into a psi
            # value.
            psiaverage = (points[i-1][1] + points[i][1])/2 + pi/2
            # Figure out which point has a higher theta value
            if points[i-1][0] < points[i][0]:

                #p_i has a higher theta value, it must be connected to
                #(2pi, psiaverage)
                x = [points[i][0]/pi, 2]
                y = [(points[i][1] + pi/2)/pi, psiaverage/pi]
                plt.plot(x, y)##################################################
                #p_i-1 has a lower theta value, it must be connected to
                #(0, psiaverage)
                x = [0, points[i-1][0]/pi]
                y = [psiaverage/pi, (points[i-1][1] + pi/2)/pi]
                plt.plot(x, y)##################################################
            else:
                #p_i has a lower theta value so it must be connected to
                #(0, psiaverage)
                x = [0, points[i][0]/pi]
                y = [psiaverage/pi, (points[i][1] + pi/2)/pi]
                plt.plot(x, y)##################################################
                #p_i-1 has a higher theta value,it must be connected to
```

```
                    #(2pi, psiaverage)
                    x = [points[i-1][0]/pi, 2]
                    y = [(points[i-1][1] + pi/2)/pi, psiaverage/pi]
                    plt.plot(x, y)################################################## 'r
            ppC.savefig(fig)
            plt.close(fig)
            gc.collect()
        ppC.close()
        print "Cylinder Plotting Complete"

## ------------------------ STADIA VIEW ------------------------------- ##
    if "b" in plotType:
        # Create a pdf for output
        ppB = PdfPages("stadia" + name + ".pdf")
        # Proceed to present the stadia view.
        fig = plt.figure(figsize=(6,4))
        x1, y1 = [-1, 1], [1, 1]
        x2, y2 = [-1, 1], [-1, -1]
        plt.plot(x1, y1, c='b')
        plt.plot(x2, y2, c='b')

        ax = plt.subplot(111)
        rightCap = mpatches.Wedge((1, 0), 1, -90, 90, width=.015, fc='b')
        leftCap = mpatches.Wedge((-1, 0), 1, 90, -90, width=.015, fc='b')

        ax.plot(0, 0, ms=0)
        ax.add_artist(rightCap)
        ax.add_artist(leftCap)

        ax.set_xlim([-2.1, 2.1])
        ax.set_ylim([-1.5, 1.5])
        ax.plot(0, 0, ms=0)
```

```
for iteration in range(start, iterations):
    fig = plt.figure(figsize=(6,4))
    x1, y1 = [-1, 1], [1, 1]
    x2, y2 = [-1, 1], [-1, -1]
    plt.plot(x1, y1, c='b')
    plt.plot(x2, y2, c='b')
    plt.suptitle("Iteration " + str(iteration), fontsize=16)

    ax = plt.subplot(111)
    rightCap = mpatches.Wedge((1, 0), 1, -90, 90, width=.015, fc='b')
    leftCap = mpatches.Wedge((-1, 0), 1, 90, -90, width=.015, fc='b')

    ax.plot(0, 0, ms=0)
    ax.add_artist(rightCap)
    ax.add_artist(leftCap)

    ax.set_xlim([-2.1, 2.1])
    ax.set_ylim([-1.5, 1.5])
    ax.plot(0, 0, ms=0)
    for sample in range(samples):
        x = [cartesianImage[sample][iteration][0], \
             cartesianImage[sample][iteration+1][0]]
        y = [cartesianImage[sample][iteration][1], \
             cartesianImage[sample][iteration+1][1]]
        print 'x = ' + str(x)
        print 'y = ' + str(y)
        plt.arrow(x[0], y[0], (x[1]-x[0]), (y[1]-y[0]), color = 'r', \
            length_includes_head = True, shape='full', head_width=.05)
    ppB.savefig(fig)
#     plt.close(fig)
#     gc.collect()
```

```
        ppB.close()
        print "Stadia Plotting Complete."

## ---------- SPHERICAL VIEW ---------- ##
    if "s" in plotType:
        # Create a sphere
        ppS = PdfPages("sphere" + name + ".pdf")
        r = 1
        pi = np.pi
        cos = np.cos
        sin = np.sin
        phi, theta = np.mgrid[0.0:pi:100j, 0.0:2.0*pi:100j]
        xxx = r*sin(phi)*cos(theta)
        yyy = r*sin(phi)*sin(theta)
        zzz = r*cos(phi)

        for iteration in range(start ,iterations):
            points = images[iteration]
            fig = plt.figure()
            ax = fig.add_subplot(111, projection='3d')

            ax.plot_surface(xxx, yyy, zzz,  rstride=1, cstride=1, color='c', \
                            alpha=0.3, linewidth=0)
            # Go through the points plotting them as we go
            for i in range(1,samples):
                # Get the theta and phi values
                theta_im1 = points[i-1][0]
                theta_i   = points[i][0]
                phi_im1   = points[i-1][1] + pi/2
                phi_i     = points[i][1] + pi/2
                # Calculate the 3d Cartesian values corresponding to the
                # theta phi values.
```

```
xx_im1     = math.sin(phi_im1)*math.cos(theta_im1)
xx_i       = math.sin(phi_i)*math.cos(theta_i)
yy_im1     = math.sin(phi_im1)*math.sin(theta_im1)
yy_i       = math.sin(phi_i)*math.sin(theta_i)
zz_im1     = math.cos(phi_im1)
zz_i       = math.cos(phi_i)


dist = abs(points[i-1][0] - points[i][0])
# See if the regular distance is shorter than the compliment of
# the distances.
if dist < (2*pi - dist):
    # "no wrapping"
    # The points must be plotted normally
    x = [xx_im1, xx_i]
    y = [yy_im1, yy_i]
    z = [zz_im1, zz_i]
    ax.plot3D(x,y,z)
else:
    # "wrapping"
    # The points must be plotted so that the line between them
    # goes the "other way" around the cylinder.
    # Get the average of the two psi values of the points.
    # Create an intermediate point to prevent wrapping it
    # has a theta value of 0.
    phiaverage = (points[i-1][1] + points[i][1])/2 + pi/2
    xx0 = math.sin(phiaverage)
    yy0 = 0
    zz0 = math.cos(phiaverage)


    # Connect p_i-1 to xx0 and plot
    x = [xx_im1, xx0]
    y = [yy_im1, yy0]
```

```
                    z = [zz_im1, zz0]
                    ax.plot3D(x,y,z)

                    # Connect p_i to xx0 and plot
                    x = [xx_i, xx0]
                    y = [yy_i, yy0]
                    z = [zz_i, zz0]
                    ax.plot3D(x,y,z)

            ax.set_xlim([-1,1])
            ax.set_ylim([-1,1])
            ax.set_zlim([-1,1])
            ax.set_aspect("equal")
            plt.tight_layout()
            ppS.savefig(fig)
            plt.close(fig)
            gc.collect()
        ppS.close()
        print "Spherical Plotting Complete"
    return 1
```

A.3 ComputeIteration.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 28 14:08:19 2018

Functions for taking a range of theta-phi values and producing images of these
sets for display. We use image in the sense of the image of a set
under a function. Uses PointSampling.py, CoordinateConversion.py, and
CollisionMap.py.

@author: Randy
"""
from PointSampling import evenSpacingSample
from PointSampling import randomSample
from CoordinateConversion import thetaphiTOxybeta
from CoordinateConversion import xybetaTOthetaphi
from CollisionMap import collisionLoop

def image_const_theta(philow, phihigh, samples, theta, iterations, lam, \
                      sampleType):
    """ low        := the lower bound of the desired range of phi values.
        high       := the upper bound of the desired range of phi values.
        num        := the number of desired phi values.
        theta      := the the theta value for each point.
        iterations := the number of iterations desired.
        lam        := the parameter that characterizes a Bunimovich Stadium.
        sampleType := specifies the sampling technique to be used.
                      'even' for evenly spaced, 'random' for uniform random.
        Uses evenly spaced samples for now... Theta is constant, only
        phi varies. The range for phi is [low, high].
    """
    if sampleType == 'even':
```

```
        # Get evenly spaced phi values
        phiArray = evenSpacingSample(philow, phihigh, samples)
    elif sampleType == 'random':
        phiArray = randomSample(philow, phihigh, samples)
    images = []
    # store the Cartesian coordinates of the trajectories
    cartesianImages = []
    # build an list of lists. The inner list is the collision points on
    # iteration i. the outer list is a list of iterations.
    for k in range(iterations):
        images.append([])
    for i in range(samples):
        phi = phiArray[i]
        # Get the Cartesian values associated with the theta-phi pair
        (x, y, beta) = thetaphiTOxybeta(theta, phi, lam)
        # Iterate collisions using (x, y, beta) as seeds.
        # This returns the trajectory of (x,y,beta)
        points = collisionLoop(x, y, beta, iterations, lam)
        # Store this trajectory in cartesianImages
        cartesianImages.append(points)
        # Convert each point in the trajectory into a theta-phi pair.
        for j in range(iterations):
            (xi, yi, betai) = points[j]
            # Convert the point and store its associated theta-phi pair as
            # the j^th collision in the trajectory of the i^th sampled point.
            # each point is stored as a theta phi pair.
#             print "(i,j) = (" + str(i) + "," + str(j) + ")"
            images[j].append(xybetaTOthetaphi(xi, yi, betai, lam))

    return images,cartesianImages

def image_const_phi(thetalow, thetahigh, samples, phi, iterations, lam, \
```

```
                    sampleType):
    """ low        := the lower bound of the desired range of theta values.
        high       := the upper bound of the desired range of theta values.
        sample     := the number of desired theta values.
        phi        := the the phi value for each point.
        iterations := the number of iterations desired.
        lam        := the parameter that characterizes a Bunimovich Stadium.
        sampleType := specifies the sampling technique to be used.
                      'e' for evenly spaced, 'u' for uniform random.
        The range for phi is [low, high].
    """
    if sampleType == 'even':
        # Get evenly spaced theta values
        thetaarray = evenSpacingSample(thetalow, thetahigh, samples)
    elif sampleType == 'random':
        thetaarray = randomSample(thetalow, thetahigh, samples)
    images = []
    # This will store the Cartesian coordinates of the trajectories
    cartesianImages = []
    # build an list of lists. The inner list is the collision points on
    # iteration i. the outer list is a list of iterations.
    for k in range(iterations):
        images.append([])
    for i in range(samples):
        theta = thetaarray[i]
        # get the Cartesian values associated with this theta-phi pair
        (x, y, beta) = thetaphiTOxybeta(theta, phi, lam)
        # Iterate collisions using (x, y, beta) as seeds.
        # This returns the trajectory of (x,y,beta)
        points = collisionLoop(x, y, beta, iterations, lam)
        # Store this trajectory in cartesianImages
        cartesianImages.append(points)
```

```
        # Convert each point in the trajectory into a theta-phi pair.
        for j in range(iterations):
            (xi, yi, betai) = points[j]
            # Convert the point and store its associated theta-phi pair as
            # the j^th collision in the trajectory of the i^th sampled point.
            # each point is stored as a theta phi pair.
            images[j].append(xybetaTOthetaphi(xi, yi, betai, lam))

    return images,cartesianImages
```

A.4 CollisionMap.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Sep 23 14:12:38 2018
Computes collisions by iterating the collision map. Implements the stadia's
Collision map to find the successive collision of a given collision. Uses
CoordinateConversion.py.

@author: Randy
"""

import math
from CoordinateConversion import mod2pi

def collisionLoop(x, y, beta, iterations, lam):
    """ Calls the collision map 'iterations' times on the initial condition
        (x,y,b). Each calculated collision is saved in a list which is returned
        by this function.
    """
    # This value sets an error tolerance when comparing two "equal" values
    global epsilon
    epsilon = math.pow(10,-7)
    global delta
    delta = math.pow(10,-6)

    # This value is used frequently in most of the functions.
    global halfLam
    halfLam = float(lam)/2

    points = [(x,y,beta)]

    # Iteratively call the collision map. Save the collision points returned.
```

```
    while iterations > 0:
        (x,y,beta) = collisionMap(x,y,beta)
        points.append((x,y,beta))
        iterations = iterations - 1

    return points


def collisionMap(x,y,beta):
    """ Calculates the next collision given the previous collision specified
        by x,y,beta. The next collision is returned as x1,y1,b1. This function
        uses the 4 boolean flags described below.
    """
    # ::The COLLISION variable::
    # This variable indicates where the previous collision was, within the
    # collision testers it is set to where the collision is. The key describing
    # the values for this variable and what they mean are as follows:
    # 0 indicates that the variable has just been reset/ collision location
    # unknown.
    # 1 indicates that the collision was with the left cap.
    # 2 indicates that the collision was with the upper side.
    # 3 indicates that the collision was with the right cap.
    # 4 indicates that the collision was with the lower side.
    # Set the variable to zero to indicate that the location of the previous
    # collision is unknown.
    collision = 0

    beta = mod2pi(beta)

    # Check for the special case where the billiard is traveling horizontally.
    if beta == 0 or beta == math.pi:
        return horizontalPath(x,y,beta)
```

```
    # Check for the special case where the billiard is traveling vertically.
    if abs(beta - math.pi/2) < delta or abs(beta - 3*math.pi/2) < delta:
        return verticalPath(x,y,beta)

    # See which part of the billiard table the initial condition (a collision)
    # resides. Then set collision to the appropriate value.
    # See which part of the table the initial condition resides.
    if x < -halfLam:
        # left cap.
        collision = 1
    elif x <= halfLam:
        if y > 0:
            # Upper side.
            collision = 2
        if y < 0:
            # Lower side.
            collision = 4
    else:
        # right cap.
        collision = 3

    if not collision == 2 :
        # Run collision test for upper side.
        point = upperSideCollisionTest(x,y,beta)
        if point != None:
            return point

    if not collision == 4:
        # Run collision test for lower side.
        point = lowerSideCollisionTest(x,y,beta)
        if point != None:
            return point
```

```
    point = leftCapCollisionTest(x,y,beta, collision)
    if point != None:
        return point

    point = rightCapCollisionTest(x,y,beta, collision)
    if point != None:
        return point

    print "The collision map did not detect a collision for the input: "
    print "[x,y,beta] = " + str([x,y,beta])
    return (0,0,0)
##########################################################END Collision Map###########


def upperSideCollisionTest(x,y,beta):
    """ This function checks for a collision with the upper side. If one is
        detected, then calculate the coordinates where it occurs and the
        billiard's new direction. The input is the coordinate of the billiard's
        previous collision along with the slope of the line representing the
        path. The coordinates of the new collision are returned along with the
        angle specifying the direction. If no collision was detected then None
        is returned.
    """
    # Set the slope of the line representing the path of the billiard.
    m = math.tan(beta)
    # See where the billiard's path intersects the line y=1.
    # If this occurs in the proper range then a collision with
    # the upper side has occurred.
    intersection = (1-float(y))/m + x
    if -halfLam <= intersection and intersection <= halfLam:
        x1 = intersection
```

```
        y1 = 1
        # find the components of the post-collisional velocity vector.
        vpx = math.cos(beta)
        vpy = -math.sin(beta)
        # use the post collisional velocity vector to find the new direction
        # angle
        beta1 = math.atan2(vpy,vpx)
        return (x1, y1, mod2pi(beta1))
    return None

##################################################END upperSideCollisionTest()####

def lowerSideCollisionTest(x,y,beta):
    """ Tests for collision with LS. IF one occurs return the new collisions
        coordinates with the billiards new direction. Otherwise return None.
    """
    # Set the slope of the line representing the path of the billiard.
    m = math.tan(beta)
    # See where the billiard's path intersects the line y=1.
    # If this occurs in the proper range then a collision with
    # the upper side has occurred.
    intersection = (-1-float(y))/m + x
    if -halfLam <= intersection and intersection <= halfLam:
        x1 = intersection
        y1 = -1
        # find the components of the post-collisional velocity vector.
        vpx = math.cos(beta)
        vpy = -math.sin(beta)
        # use the post collisional velocity vector to find the new direction
        # angle
        beta1 = math.atan2(vpy,vpx)
        return (x1, y1, mod2pi(beta1))
```

```
    return None

##################################################END lowerSideCollisoinTest()######

def leftCapCollisionTest(x,y,beta,collision):
    """ Tests for a collision with the upper left cap. If one is detected,
        the new x,y, and beta-values are computed and returned. If not, then
        None is returned.
    """
    # Set the slope of the line representing the path of the billiard.
    m = math.tan(beta)
    # Set variables used to test for a collision
    mSquare = math.pow(m,2)
    a = mSquare + 1
    b = 2*m*y - 2*mSquare*x + 2*halfLam
    c = mSquare*math.pow(x,2) - 2*m*y*x + math.pow(y,2) \
        + math.pow(halfLam,2) - 1
    bSquare = math.pow(b,2)
    discr = bSquare - 4*a*c
    # Ensure that the discriminant is non-negative!
    if discr < 0:
        return None
    #
    square = math.sqrt(discr)
    a2 = 2*a
    # Find the two potential x-values of a collision with the upper left cap.
    d = (-b + square)/a2
    e = (-b - square)/a2
    # See if the previous collision was in the left cap.
    if collision == 1 :
        # The previous collision was in the left cap. See which
        # potential x-value was the x-value of the previous collision.
```

```
        # The other potential x-value must belong to the new collision.
        # Set collision to 1  to indicate that a collision with left cap
        # was detected so that the new y-value and direction can be computed
        # below.
        if abs(x-e) < epsilon and d < -halfLam:
            x1 = d
            collision = 1

        elif abs(x-d) < epsilon and e < -halfLam:
            x1 = e
            collision = 1
        else:
            # The previous collision was in the upper left cap and the
            # old x-value does not correspond to one of the solutions of the
            # billiard's path intersecting the cap.
            collision = 0
    else:
        # The previous collision was not in the upper left cap. Test potential
        # x-values to see if they lie on the upper left cap.
        if -halfLam -1 <= d and d < -halfLam:
            # A collision was detected, set the new x-value, collision to 1
            # so the new y-value and direction can be computed below.
            x1 = d
            collision = 1
        elif -halfLam -1 <= e and e < -halfLam:
            # A collision was detected, set the new x-value, collision to 1
            # so the new y-value and direction can be computed below.
            x1 = e
            collision = 1
    # If a collision was detected we must compute the new y-value and direction
    # Otherwise we return None as specified.
    if collision == 1:
```

```
            # Set the new y-value
            y1 = m*(x1 - x) + y
            # Calculate the components of the pre-collisional velocity vector
            vmx = math.cos(beta)
            vmy = math.sin(beta)
            # Calculate the components of the inward normal vector
            nx = -(x1 + halfLam)
            ny = -y1
            # calculate the dot product of the pre-collisional velocity vector and
            # the inward normal
            dot = vmx*nx + vmy*ny
            # Calculate the components of the post-collisional velocity vector
            vpx = vmx - 2*dot*nx
            vpy = vmy - 2*dot*ny
            # Calculate the direction angle of the post-collisional velocity vector
            beta1 = math.atan2(vpy, vpx)
            return (x1, y1, mod2pi(beta1))
    return None

########################################End Left Cap Collision Tester

def rightCapCollisionTest(x,y,beta, collision):
    """ Tests for a collision with the upper right cap. If one is detected,
        then the new x,y, and beta-values are computed and returned. If not,
        then None is returned.
    """
    # Set the slope of the line representing the path of the billiard.
    m = math.tan(beta)
    # Set variables used to test for a collision
    mSquare = math.pow(m,2)
    a = mSquare + 1
    b = 2*m*y - 2*mSquare*x - 2*halfLam
```

```
c = mSquare*math.pow(x,2) - 2*m*y*x + math.pow(y,2) \
    + math.pow(halfLam,2) - 1
bSquare = math.pow(b,2)
discr = bSquare - 4*a*c
# Ensure that the discriminant is non-negative!
if discr < 0:
    return None
square = math.sqrt(discr)
a2 = 2*a
# Find the two potential x-values of a collision with the upper right cap.
d = (-b + square)/a2
e = (-b - square)/a2
# Check if the previous collision was with the upper right cap.
if collision == 3:
    # The previous collision was in the upper right cap. See which
    # potential x-value was the x-value of the previous collision.
    # The other potential x-value must belong to the new collision.
    # Set collision to 3 to indicate that a collision with upper right cap
    # was detected so that the new y-value and direction can be computed
    # below.
    if abs(x-d) < epsilon and e > halfLam:
        x1 = e
        collision = 3
    elif abs(x-e) < epsilon and d > halfLam:
        x1 = d
        collision = 3
    else:
        # The previous collision was in the upper right cap and the
        # old x-value does not correspond to one of the solutions of the
        # billiard's path intersecting the cap.
        collision = 0
else:
```

```
            # The previous collision was not in the right cap. Test the potential
            # x-values to see if they lie on the right cap.
            if halfLam <= d and d <= halfLam + 1 + epsilon:
                # A collision was detected, set the new x-value, collision to 3
                # so the new y-value and direction can be computed below.
                x1 = d
                collision = 3
            elif halfLam <= e and e <= halfLam + 1 + epsilon:
                # A collision was detected, set the new x-value, collision to 3
                # so the new y-value and direction can be computed below.
                x1 = e
                collision = 3
        # If a collision was detected we must compute the new y-value and direction
        # Otherwise we return None as specified.
        if collision == 3:
# Set the new y-value
            y1 = m*(x1 - x) + y
            # Calculate the components of the pre-collisional velocity vector
            vmx = math.cos(beta)
            vmy = math.sin(beta)
            # Calculate the components of the inward normal vector
            nx = -(x1 - halfLam)
            ny = -y1
            # calculate the dot product of the pre-collisional velocity vector and
            # the inward normal
            dot = vmx*nx + vmy*ny
            # Calculate the components of the post-collisional velocity vector
            vpx = vmx - 2*dot*nx
            vpy = vmy - 2*dot*ny
            # Calculate the direction angle of the post-collisional velocity vector
            beta1 = math.atan2(vpy, vpx)
            return (x1, y1, mod2pi(beta1))
```

```
    return None

########################################End Right Cap Collision Tester


def horizontalPath(x,y,beta):
    """ This function calculates the next collision for a particle traveling
        in a purely horizontal direction. (the set where this happens should
        have a measure of zero)
    """
    # The billiard is traveling horizontally so the next collision has the
    # same y-value and the x-value is reflected about the y-axis.
    y1 = y
    x1 = -x

    # Check for the special case where the billiard is traveling along the
    # x-axis. If it is it shall bounce back along the same path.
    if y == 0:
        if beta == 0:
            beta1 = math.pi
        elif beta == math.pi:
            beta1 = 0
        return (x1, y1, mod2pi(beta1))

    # Check if the billiard is traveling from left to right.
    # Notice that the billiard can not collide with a flat wall because
    # it is traveling horizontally, which is parallel to the flat walls.
    if beta == 0:
        # This is the angle between the billiard's path and the inward normal.
        # Since beta is zero, the direction between the path's line and the
        # inward normal is just the inward normal's angle.
        alpha = math.acos(x1 - halfLam)
```

```
        # Check if the billiard is traveling towards the upper right cap.
        if y > 0:
            # To get the new direction of the billiard we use the
            # direction of the inward normal and rotate it ccw by alpha.
            beta1 = math.pi + math.asin(y1) + alpha
        # Check if the billiard is traveling towards the lower right cap.
        elif y < 0:
            # To get the new direction of the billiard we use the direction
            # of the inward normal and rotate it clockwise by alpha.
            beta1 = math.pi + math.asin(y1) - alpha
        # Notice that we don't have to consider the case where y=0 because it
        # is handled in the first if statement of this function.
        # Return the new point in the collision space.
        return (x1, y1, mod2pi(beta1))

    # Check if the billiard is traveling from right to left.
    if beta == math.pi:
        # Find the angle between the billiard's path and the inward normal.
        alpha = math.acos(-(x1 + halfLam))
        # Check if the billiard is traveling towards the upper left cap.
        if y > 0:
            # To get the new direction of the billiard take the inward normal's
            # direction and rotate if clockwise by alpha.
            beta1 = 2*math.pi - math.asin(y1) - alpha
        # Check if the billiard is traveling towards the lower left cap.
        if y < 0:
            beta1 = -math.asin(y1) + alpha
        # Return the new point in the collision space.
        return (x1, y1, mod2pi(beta1))
##############################################################END horizontalPath
```

```
def verticalPath(x,y,beta):
    """ This function calculates the next collision for a particle traveling
        in a purely vertical direction.
    """
    # The x-value will remain unchanged.
    x1 = x
    y1 = -y
    # See if the billiard is in the left cap.
    if x < -halfLam:
        # See if the billiard is moving up.
        if abs(beta - math.pi/2) < delta:
            # This gives the polar angle with respect to the circle's origin
            # of the new collision point.
            thetahat = math.acos(x1 + halfLam)
            # This gives the new direction of the billiard. This formula was
            # derived using basic geometry and the fact that the previous path
            # was parallel with the y-axis (which allowed for simplifications)
            beta1 = mod2pi(math.pi + 2*thetahat)
        # See if the billiard is moving down
        if abs(beta == 3*math.pi/2) < delta:
            thetahat = 2*math.pi - math.acos(x1 + halfLam)
            beta1 = mod2pi(2*thetahat - 5*math.pi/2)
        return (x1, y1, mod2pi(beta1))

    # See if the billiard is traveling between flat walls
    if x <= halfLam:
        # See if the billiard is traveling towards the upper side. If so it
        # will just be reflected back to where it came from.
        if abs(beta - math.pi/2) < delta:
            beta1 = 3*math.pi/2
        else:
            beta1 = math.pi/2
```

```
        return (x1, y1, mod2pi(beta1))

    # See if the billiard is in the right cap
    else:
        if abs(beta - math.pi/2) < delta:
            thetahat = math.acos(x1 - halfLam)
            beta1 = mod2pi(math.pi/2 + 2*thetahat)
        elif abs(beta == 3*math.pi/2) < delta:
            thetahat = 2*math.pi - math.acos(x1 - halfLam)
            beta1 = mod2pi(2*thetahat - 5*math.pi/2)
        return (x1, y1, mod2pi(beta1))

##########################################################################END verticalPath
```

A.5 CoordinateConversion.py

```
# -*- coding: utf-8 -*-
"""
Created on Sat Oct 27 16:42:30 2018
Provides functions for conversion of points in the collision space of the
Bunimovich stadium.
This provides a library of functions that transform points in the collision
space to and from Cartesian and spherical.

@author: Randy
"""
import math

def thetaphiTOxybeta(theta, phi, lam):
    """ theta := polar angle of the location where a collision occurs.
        phi   := angle of the post-collisional velocity vector WRT the inward
                 normal vector.
        lam   := the ratio of twice the length of the billiard table to its
                 radius.
        This function takes (theta, phi) and produces the x,y pair representing
        that location in Cartesian coordinates. It also finds beta, the polar
        angle of the direction of the velocity vector.
    """
    # Calculate the reference angle associated with this instance of a
    # Bunimovich Stadium. This angle will be used to determine on which wall
    # the point resides.
    ref = math.atan(float(2)/lam)
    halfLam = float(lam)/2
    theta = mod2pi(theta)

    # Note: if lam = 2 then ref = 0.785398163397 = pi/4
    # See if the point is on the top or bottom side.
```

```
# Top side
if  theta >= ref and theta <= math.pi - ref:
     x = float(1)/math.tan(theta)
     y = 1
     beta = 3*math.pi/2 + phi
     return (x, y, beta)

# Bottom side
if math.pi + ref <= theta and theta <= 2*math.pi - ref:
    x = -float(1)/math.tan(theta)
    y = -1
    beta = math.pi/2 + phi
    return (x, y, beta)

# Calculate values needed for locations in either cap.
cos = math.cos(theta)
sin = math.sin(theta)

# Left cap
if math.pi - ref < theta and math.pi + ref > theta:
    rho = -halfLam*cos + math.sqrt(1 - math.pow(sin*halfLam,2))
    x   = rho*cos
    y   = rho*sin
    vx = -(x + halfLam)*math.cos(phi) + y*math.sin(phi)
    vy = -(x + halfLam)*math.sin(phi) - y*math.cos(phi)
    beta = mod2pi(math.atan2(vy,vx))
    return (x, y, beta)

# Right cap
if (0 <= theta and theta < ref) or \
(2*math.pi - ref < theta and theta <= 2*math.pi):
    rho = halfLam*cos + math.sqrt(1 - math.pow(sin*halfLam,2))
```

```
        x   = rho*cos
        y   = rho*sin
        vx = -(x - halfLam)*math.cos(phi) + y*math.sin(phi)
        vy = -(x - halfLam)*math.sin(phi) - y*math.cos(phi)
        beta = mod2pi(math.atan2(vy,vx))
        return (x, y, beta)
#------------------------------------------------- End of thetaphiTOxybeta


#  (x,y, beta) |-----> (theta, phi)
def xybetaTOthetaphi(x, y, beta, lam):
    """ (x,y) := The Cartesian coordinates of the collision.
        beta  := The polar angle of the velocity vector.
        This function takes a (x,y,beta) triple and produces a
        theta phi pair.
    """
    # Calculate theta
    theta = mod2pi(math.atan2(y, x))
    # This value is commonly used...
    halfLam = float(lam)/2

    # We shall calculate psi \in [0, pi]. This is the angle between the
    # post collisional velocity vector and the positively oriented tangent to
    # the stadium's boundary. We calculate psi to find phi.
    # Right cap
    if x > halfLam:
        # switched sign of y*cos(beta) and the second term too.
        psi = math.acos(-y*math.cos(beta) + (x - halfLam)*math.sin(beta))
    # The Sides
    elif x >= -halfLam:
        # Top Side
        if y == 1:
```

```
            psi = beta - math.pi
        elif y == -1:
            psi = beta
        else:
            # If the x-coordinate is between -halfLambda and halfLambda
            # inclusive then the y-coordinate must be 1 or -1 because
            # the billiard must have intersected one of the flat walls.
            # So if y is neither of these values an error message should be
            # displayed.
            print "An attempt was made to convert x,y to theta,phi but"
            print "the x-value corresponds to a side of the stadia and"
            print "the y-value does not! The bad values are: "
            print "[x,y,beta] = " + str([x,y,beta])
    # Left cap
    else:
        psi = math.acos(-y*math.cos(beta) + (x + halfLam)*math.sin(beta))
    # Use psi to find phi
    phi = psi - math.pi/2
    return (theta, phi)
#------------------------------------------------------------ End xybetaTOthetaphi


def mod2pi(angle):
    """ Takes an angle and returns the co-terminal angle in [0,2pi).
    """
    while angle < 0:
        angle = angle + 2*math.pi
    while angle >= 2*math.pi:
        angle = angle - 2*math.pi
    return angle
```

A.6 PointSampling.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 28 12:58:30 2018

Handles sampling given numerical ranges.

@author: Randy
"""
import numpy as np

def evenSpacingSample(low, high, num):
    """ Samples evenly spaced points in the interval [low, high].
        Returns num number of evenly spaced points in [low, high].
        USES NUMPY.
    """
    return np.arange(low, high, (high-low)/float(num)).tolist()

def randomSample(low, high, num):
    """ 'Randomly' samples points in the interval [low, high].
        Returns num # of randomly sampled points in [low, high].
        USES NUMPY
    """
    # generate num random numbers between 0 and 1.
    sample = np.random.rand(num)
    # find the range of numbers
    rangee = high - low
    # scale every number in the array by rangee
    sample = rangee*sample
    # shift the array of numbers into the interval [low, high]
    sample = sample + low
    sample = sample.tolist()
```

```
    sample.sort()
    return sample
```

Appendix B

Instructions for using the Bunimovich Stadia Evolution Viewer Software

B.1 Using the GUI

1. To use the Bunimovich Stadium Evolution Viewer you must first run the *BunimovichStadiumViewer-GUI.py* file. If you are using windows 10 you can do this by clicking on the file named *BSE GUI.exe*. Once the software is running the window in Figure B.1 should pop up.

2. To use the software fill in all of the fields and click the *Generate* button to produce pdf(s). Please note that all fields must have a value and atleast one of the check boxes must be checked before *Generate* is clickable. For an explanation of all of the fields and the software please see below.

Explanation of the Software

This software allows the user to see how the orbits of nearby billiards in the Bunimovich stadium evolve over time. A set of nearby points in the collision space is the input of the program. The output is visual data representing the successive collisions of the points in the input set. Recall that in the Bunimovich stadia the collision space is represented by points of the form (θ, φ). θ is the polar angle of the collision point on the stadium's boundary. φ is the angle between the inward normal $\hat{N}$ of the stadium at θ and the post-collision velocity vector. The two parameters have the following ranges $\theta \in [0, 2\pi)$ and $\varphi \in (-\frac{\pi}{2}, \frac{\pi}{2})$. Graphical representations of θ and φ can be seen in Figure B.2.

The input sets can be of two different forms:

(1) A set of collision points with θ constant and φ varying, or

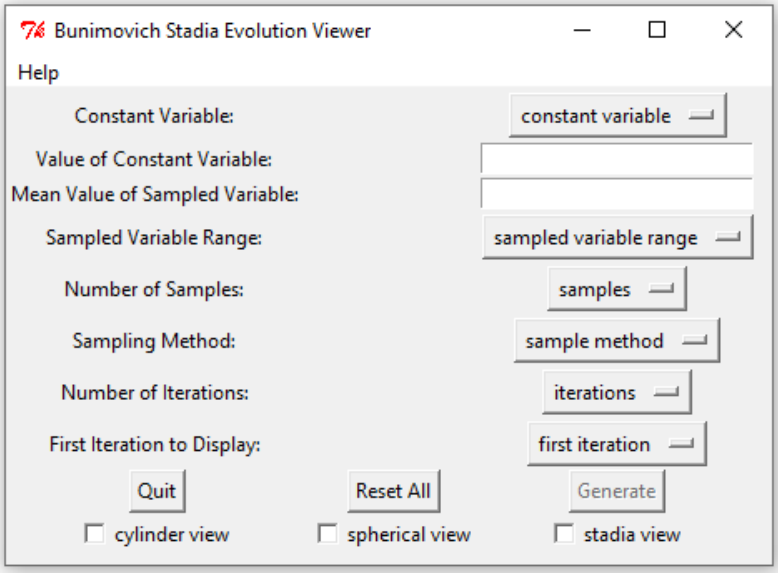

Figure B.1: The graphical user interface (GUI) for the Bunimovich Stadium Evolution Viewer Software

(2) A set of collision points with φ constant and θ varying.

Once the inputs are specified the software can produce 3 different kinds of views of the system's evolution over successive applications of Bunimovich stadium's collision map. The first view is the trajectory of the billiards corresponding to the input set "bouncing" in the stadia. To understand the second view notice that the collision space is homeomorphic to a cylinder since θ is cyclic and φ is in an interval. Thus, the second view shows the billiards' trajectories represented on the cylinder[1]. The third view is the collision space projected on a sphere. Once the inputs have been supplied and the *Generate* button is clicked, pdf(s) of the desired views are produced in the directory containing the .py or .exe file.

[1]The cylinder is "cut" and "flattened out" so it can be easily viewed on a flat screen.

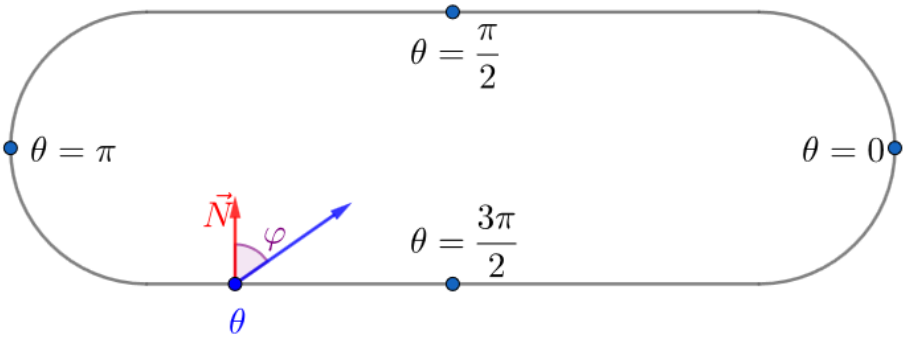

Figure B.2: The inward normal $\hat{N}$, φ, and θ

Providing Input to the Software

There are eight fields of input that must be supplied to the software in addition to the check boxes that specify which views to produce. For an explanation of a particular input field please find its label below.

Constant Variable

This specifies which variable is to be constant, either θ (theta) or φ (phi).

Value of Constant Variable

This sets the value of the constant variable. To supply the proper input recall that $\theta \in [0, 2\pi)$ and $\varphi \in (-\frac{\pi}{2}, \frac{\pi}{2})$.

Mean Value of Sampled Variable

This, along with *Sampled variable Range* specifies the range of values that the non-constant variable is allowed to take. For instance if 1 is specified as the *Mean Value of Sampled Variable* and "pi/8", (about 0.392) is specified as *Sampled variable Range* then the range of the non-constant variable would be (about) 1 ± 0.392.

Sampled Variable Range

(Please see the above explanation of *Mean Value of Sampled Variable*).

Number of Samples

The software can not compute the trajectory of every point in the ranges specified by the user. Instead it takes a sample of points in the range. This field tells the software how many points to sample. The more points that are sampled, the more accurate the output is.

Sampling Method

This specifies the sampling technique to be used by the software. If "even" is selected then the points are evenly spaced in the non-constant variable's range. If "random" is selected then points in the specified range are chosen according the the uniform distribution.

Number of Iterations

This specifies the number of iterations of the collision map to be applied to the sampled points.

First iteration to Display

This specifies the first iteration to be displayed in the output pdf. For example if *Number of Iterations* is set to "10" and *First iteration to Display* is set to "5" then iterations 5-10 of the collision map will be displayed in the pdf(s). Please note that this value must be less than the value of *Number of Iterations*.

Checkboxes

The check boxes specify which views will be displayed in the pdf(s). At least one of these must be checked, however the user can check as many as is desired. A pdf is created for each box checked in the same directory as the .exe or .py file.

Bibliography

[1] Adams, C.C. and Franzosa, R.D. *Introduction to Topology: Pure and Applied*. Pearson Prentice Hall, 2008. ISBN: 9780131848696. URL: https://books.google.com/books?id=W1wnAQAAIAAJ (cit. on pp. 23, 25, 37).

[2] Baez, John. *Bunimovich Stadium — Visual Insight*. 2016. URL: https://blogs.ams.org/visualinsight/2016/11/15/bunimovich-stadium/ (visited on 03/06/2019) (cit. on pp. 14, 16, 39, 59).

[3] Bunimovich, Leonid A. "On ergodic properties of certain billiards". In: *Functional Analysis and Its Applications* 8.3 (1974), pp. 254–255 (cit. on pp. 9, 16, 26, 27).

[4] Bunimovich, Leonid A. "On the ergodic properties of nowhere dispersing billiards". In: *Communications in Mathematical Physics* 65.3 (1979), pp. 295–312 (cit. on pp. 9, 26, 59).

[5] Chernov, N. and Markarian, R. *Chaotic Billiards*. Mathematical surveys and monographs. American Mathematical Society, 2006. ISBN: 9780821840962. URL: https://books.google.com/books?id=Zu5u7wfHstsC (cit. on pp. 9, 11, 16, 17, 20, 26, 27, 28, 36, 37, 82).

[6] Devaney, R.L. *A First Course In Chaotic Dynamical Systems: Theory And Experiment*. Studies in Nonlinearity. Avalon Publishing, 1992. ISBN: 9780813345475. URL: https://books.google.com/books?id=OkYHSu-dEEYC (cit. on p. 24).

[7] Dmcq, Wikimedia Commons User. *Spherical Coordinate System in Mathematics*. File: 3D Spherical 2.svg. 2012. URL: https://en.wikipedia.org/wiki/File:3D_Spherical_2.svg (visited on 03/06/2019) (cit. on p. 32).

[8] Nillsen, R. and America, Mathematical Association of. *Randomness and Recurrence in Dynamical Systems: A Real Analysis Approach*. Carus Mathematical Monographs. Mathematical Association of America, 2010. ISBN: 9780883850435. URL: https://books.google.com/books?id=NkzPSr-JpBwC (cit. on p. 59).

[9] Rozenbaum, Efim, Ganeshan, Sriram, and Galitski, Victor. "Characterization of Classical and Quantum Chaos With Out-of-Time-Ordered Correlator in Stadium Billiard". In: *APS Meeting Abstracts*. 2018 (cit. on pp. 9, 14).

[10] Scheidegger, Carlos. *Bunimovich Stadium*. 2018. URL: `https://cscheid.net/projects/bunimovich_stadium/` (visited on 03/06/2019) (cit. on pp. 14, 16, 39).

[11] Sinai, Yakov Grigor'evich. "Dynamical systems with elastic reflections". In: *Russian Mathematical Surveys* 25.2 (1970), pp. 137–189 (cit. on pp. 16, 26, 27).

[12] Weaver, Bryce. "Growth Rate of Periodic Orbits for Geodesic Flows Over Surfaces with Radially Symmetric Focusing Caps." In: *Journal of Modern Dynamics* 8.2 (2014) (cit. on p. 33).